GREP.C

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

Dr. Dobb's Journal

#96 October 1984

$2.95 (3.50 Canada)

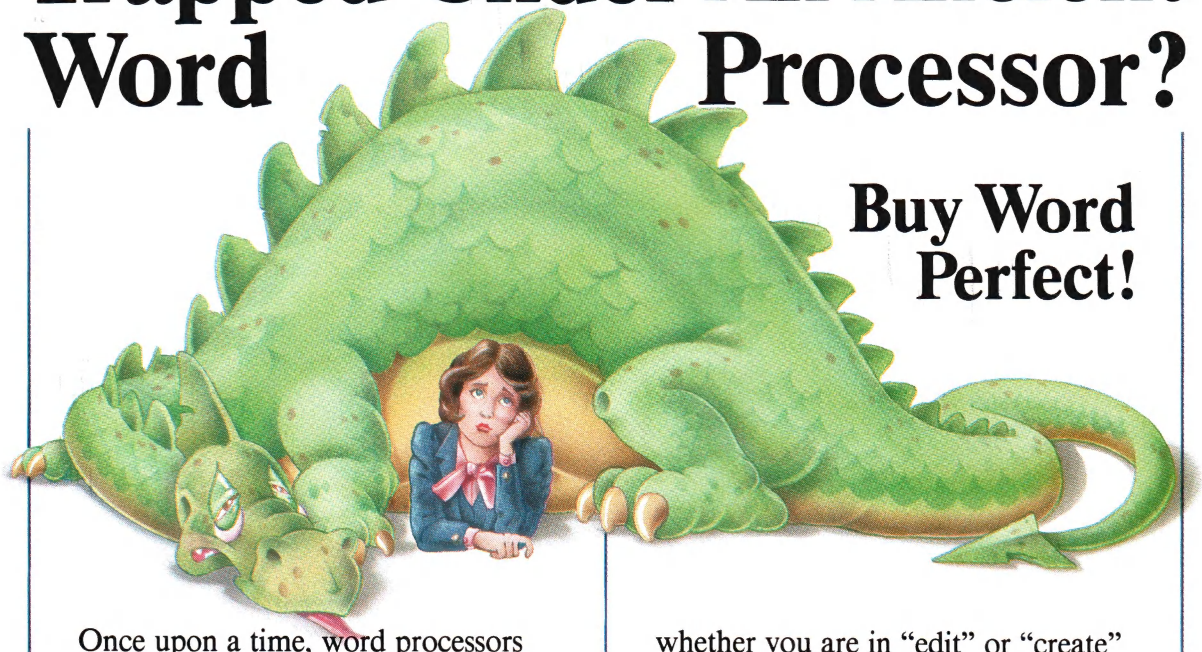
New Column: The Software Designer

East Coast vs West Coast Programming Style

Simple Calculations With Complex Numbers

More dBASE II Programming Techniques

How Can You Avoid Getting Trapped Under An Ancient Word Processor?



Buy Word Perfect!

Once upon a time, word processors were monstrous things. Dot commands, page orientation, and separate editing, formatting and printing programs turned them into lumbering beasts. Only a well-educated programmer would dare don his armor and tackle such a beast — not a pleasant task for a modern secretary, executive, or writer.

Then came WordPerfect and the beast was slain.

WordPerfect was designed to work for you not against you. WordPerfect has no command language to complicate your writing. Pressing a single key is all it takes to

bold, underline or center.

When writing, you don't want to worry about page formatting, making room for headers or footers, or whether you are in "edit" or "create" mode. Your word processor should do it automatically and WordPerfect does. WordPerfect lets you think in terms of ideas, not pages. It is simple enough that you quickly forget about the mechanics and your writing flows easily.

So if you don't want to be caught under a word processing monster, try WordPerfect. We're certain it will improve the quality of your writing.

WordPerfect

You'll love it — not only for the features we've built in, but also for the antiquities we've left out.

SATELLITE SOFTWARE INTERNATIONAL

288 WEST CENTER STREET, OREM, UTAH 84057 (801) 224-4000 TELEX 820-618

SSI

Circle **no. 60** on reader service card.

SCREEN SCULPTOR

GENERATES CUSTOMIZED INPUT SCREEN PROGRAMS IN BASIC AND PASCAL . . .

. . . BECAUSE YOUR TIME IS TOO VALUABLE

Only **$125^{00}**

ITEM #2010

(Includes shipping and handling)
(N.Y.S. Res. Add 8¼% sales tax)

BASIC or PASCAL. Easily!

Generate programs in **BASIC** or **PASCAL**. Your **choice.**
Works with **Turbo PASCAL * IBM * PASCAL** or **IBM BASIC** (Interpreter and Compiler).
Easy to use. Begin productive use in minutes!

POWERFUL SCREENS

Everyone can have **professional quality screens** to dress up any program.
Generate **complex, colorful, effective** screens in minutes.

FULL FUNCTION SCREEN CREATION

Simply **"draw" input screens** with word processor style editor.

Advance screen creating features include:
- **Draw boxes, lines, etc.** in seconds with unqiue character selection menu.
- **Repeat last character** in any direction.
- Special **color-select screen** displays all available colors.
- **Paint and Repaint** sections of screen at any time.
- **Copy** and **Move** sections of screen.
- **Insert** or **Delete characters** and **lines.**
- **Input field definition** screen gives you **total control** of character type definitions, edit screen masks, input sequence, variable names, initial values, protected characters, etc.

COMPLETE DATA ENTRY ROUTINES

Generates customized program code that allows **professional quality** data input using the full PC keyboard **(cursor keys, delete, insert, and more). Checks input data** for valid entries and **displays error messages.**
Programs are easily **merged** with your own programs.
Easy to follow documentation shows how and where you can **modify the generated programs.**

- **Available now with IBM PC, PCjr, PCXT, and all true compatibles.**
- **Requires 128k RAM, one floppy disk drive, and PC DOS. Works with any display type.**

* Turbo PASCAL is a registered trademark of Boreland International, Ltd. IBM is a registered trademark of IBM corporation.

Master Card and **Visa** orders call now *toll free* 1-800-824-7888. Operator 268
Alaska and Hawaii call 1-800-824-7919. Operator 268
DEALER INQUIRIES INVITED. SORRY, NO C.O.D.
Produced and Distributed by **The Software Bottling Co. of New York.**
29-14 23 Ave., New York City, N.Y. 11105 • (212) 728-2200 Telex 380748


dBASE III™
dBASE III™
SYSTEM DISK
Serial No: 100001
Format: IBM PC, XT and 100% compatibles PC-DOS 2.0
Version: 1.0
ASHTON·TATE
10150 West Jefferson Blvd.
Culver City, CA 90230 (213) 204-5570
TELEX 664228 ASHTATE CULV
©Copyright 1984 Ashton-Tate as an unpublished licensed proprietary work.
All rights reserved.

More power to you.

Remember the magic you expected when you first purchased a PC?

It's here.

dBASE III™ is the most powerful database management system ever created for 16-bit microcomputers. It pulls every ounce of energy out of your PC and puts it to work.

On top of that, it's fast and it's easy.

You've never seen anything like it.

dBASE III can handle over a billion records per file, limited only by your computer system. You can have up to ten files open, for sophisticated applications programs.

When you have two related files, information in one can be accessed based upon data in the other.

dBASE III now handles procedures, parameter passing and automatic variables. You can include up to 32 procedures in a single file. With lightning speed. Because once a file is opened, it stays open. And procedures are accessed directly.

Easier than ever.

dBASE III uses powerful yet simple commands that are the next best thing to speaking English.

If you're unsure of a command, HELP will tell you what to ask for.

If you don't know what command comes next, a command assistant does. All you have to know is what you want it to do.

Our new tutorial/manual will have you entering and viewing data in minutes rather than reading for hours.

And to make matters easier, you get a full screen report setup for simple information access.

Faster than no time at all.

dBASE III isn't just fast. It's ultra-fast. Operating. And sorting. Even faster, is no sorting. Because dBASE III keeps your records in order, so you really don't have to sort anything. Unless you want to. Then watch out!

What about dBASE II®?

It's still the world's best database management system for 8-bit computers. And it's still the industry standard for accounting, educational, scientific, financial, business and personal applications.

Tap into our power.

For the name of your nearest authorized dBASE III dealer, contact Ashton-Tate, 10150 West Jefferson Boulevard, Culver City, CA 90230. (800) 437-4329, ext. 333. In Colorado, (303) 799-4900.

ASHTON·TATE™

Dr. Dobb's Journal

Editorial
Editor-in-Chief *Michael Swaine*
Editor *Reynold Wiggins*
Managing Editor *Randy Sutherland*
Contributing Editors *Robert Blum, Dave Cortesi, Ray Duncan, Anthony Skjellum, Michael Wiesenberg*
Copy Editors *Polly Koch, Cindy Martin*
Typesetter *Jean Aring*
Editorial Intern *Mark Johnson*

Production
Design Director *Fred Fehlau*
Art Director *Shelley Rae Doeden*
Production Manager *Detta Penna*
Production Assistant *Alida Hinton*
Cover *William Cone*

Advertising
Advertising Sales *Alice Hinton, Walter Andrzejewski*

Circulation
Circulation and Promotions Director *Beatrice Blatteis*
Fulfillment Manager *Stephanie Barber*
Direct Response Coordinator *Maureen Snee*
Promotions Coordinator *Jane Sharninghouse*
Single Copy Sales Coordinator *Sally Brenton*
Single Copy Sales *Lorraine McLaughlin*
Circulation Assistant *Kathleen Boyd*

M&T Publishing, Inc.
Chairman of the Board *Otmar Weber*
Director *C.F. von Quadt*
President *Laird Foshay*

Dr. Dobb's Journal (USPS 307690) is published monthly by M&T Publishing, Inc., 2464 Embarcadero Way, Palo Alto, CA 94303, (415) 424-0600. Second class postage paid at Palo Alto and at additional entry points.

Address correction requested. Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, 2464 Embarcadero Way, Palo Alto, CA 94303. **ISSN 0278-6508**

Subscription Rates: $25 per year within the United States, $44 for first class to Canada and Mexico, $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Contributing Subscribers: Christine Bell, W.D. Rausch, DeWitt S. Brown, Burks A. Smith, Robert C. Luckey, Transdata Corp., Mark Ketter, Friden Mailing Equipment, Frank Lawyer, Rodney Black, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor Jr., Ted A. Reuss III, InfoWorld, Stan Veit, Western Material Control, S.P. Kennedy, John Hatch, Richard Jorgensen, John Boak, Bill Spees, R.B. Sutton. **Lifetime Subscribers:** Michael S. Zick, F. Kirk.

Foreign Distributors: ASCII Publishing, Inc. (Japan), Computer Services (Australia), Computer Store (New Zealand), Computercollectief (Nederland), Homecomputer Vertriebs GMBH (West Germany), International Presse (West Germany), La Nacelle Bookstore (France), McGill's News Agency PTY LTD (Australia), Progresco (France).

People's Computer Company
Dr. Dobb's Journal is published by M&T Publishing, Inc. under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a non-profit, educational corporation.

OCTOBER 1984
VOLUME 9, ISSUE 10

CONTENTS

This Month

If you have not already noticed, we have a new column starting this month. Mike Swaine will be writing The Software Designer, a column intended to present the views of industry leaders on software design topics. This debut discusses issues of programming style differences. Tools for software design will be next month's topic.

Also as of this issue, Of Interest will be written by Randy Sutherland. Our congratulations to Randy on his new endeavor. Michael Wiesenberg, who has handled that task for nearly two and a half years, will be moving on to do other writing for us, including one of the new puzzle columns we have coming up. Our thanks to Michael for the job he's done for us. We look forward to his future contributions.

Coming Down the Pike

We've been looking at our editorial calendar for 1985 and thought we'd let you in on some of the special issues we've planned. As you already know, we plan to focus on Unix this December. The February 1985 issue will be our 100th issue, and we figure we should do some celebrating. March 1985 will be an artificial intelligence issue, including the winner from the Fifth Generation Programming Competition (you did get your entry in, didn't you?). June 1985 will focus on telecommunications. September will still be the annual Forth issue, and we will close out the year with a focus on the latest developments in operating systems (the exact topics, of course, partially dictated by what transpires during the year).

While this is only part of what we have in store, it will give readers something to look forward to and authors something to plan toward. Authors should note that the copy deadline is likely to be about three months prior to the issue date, though we will announce deadlines as we get closer to the issues.

This Month's Referees

Dr. Dobb's Journal regularly draws on the expertise of a Board of Referees for technical evaluation of material submitted for publication. In addition to remarks to the editors concerning accuracy and relevance of manuscripts, the referees often provide constructive comments for authors regarding clarity or completeness. Their remarks help prevent authors form exposing blindspots or misconceptions in print and help ensure that our readers receive clear and accurate information. The referees who contributed to this month's issue are:

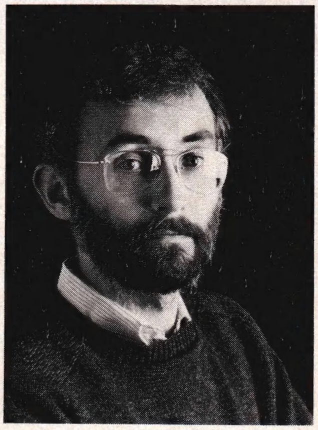
Wayne Chin, Hewlett-Packard, Information Networks Division
Georges Grinstein, Computer Science Dept., Fitchburg State College
John P. Keyes, Microsoft
Ben Laws, Computer Science, North Texas State University
Scott D. Thomas, Informatics General Corporation

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

Dr. Dobb's Journal

ARTICLES

DEPARTMENTS



Five months ago I wrote my first editorial for *DDJ* and in it made several promises about the editorial direction of the magazine. Since then, editor Reynold Wiggins and I have spent twenty-odd weeks examining the Doctor, and recently the entire *DDJ* staff spent a three-day weekend poking *DDJ* with every available instrument. The prognosis is excellent; *DDJ* is a healthy, growing magazine, and I can now be more specific about what the magazine will not be doing over the next year.

DDJ will not abandon 8-bit software so long as useful creative software development is being done on 8-bit systems; forever, so far as we can tell. *DDJ* will, on the other hand, focus more deliberate attention on 16- and 32-bit software. We'll publish MSDOS- and Unix-based software and we'll investigate software development on the newer microprocessors. If you conclude that we intend to stay pretty loose, you're right.

DDJ will not become a product-specific magazine. We'll always prefer the MSDOS application to the PCDOS and the DOS-independent application to the strictly MSDOS. We are interested in the Mac/Lisa/PARC human interface, naturally; it's a highly significant development in personal computer software. But the insights embodied in the Mac are not proprietary.

DDJ will not become language-specific, either. Most of the code published in other computer magazines is in BASIC. Not so in *DDJ*, since we try to furnish consequential code for software designers. Despite the fact that this magazine was founded to put a version of BASIC in the public domain, its editors have never considered BASIC an ideal language either for software development or for communicating ideas about programming.

Apparently Kemeney and Kurtz agree; the designers of BASIC seem to have concluded that BASIC would be better off being Pascal (i.e. True BASIC) just as Pascal's designer declares his language obsolete, to be succeeded by Modula-2. Pascal remains, however, a good notation for communicating algorithms, so we'll doubtless continue to publish some Pascal code. We'll be investigating Modula-2, but we do have some questions about how and in what settings it's likely to be useful (see "The Software Designer" in this issue). Our Modula coverage is likely to focus at least for the immediate future on what the language has to offer programmers (as in "An Introduction to Modula-2 for Pascal Programmers," May 1984), rather than on providing tools for Modula programmers.

DDJ has participated in the evolution of Forth as a language, but we will shift our coverage of Forth slightly over the next few months. We disagree with *Byte* magazine's assessment that Forth is still a language in flux. Such developments as the FVG Forth floating-point standard (described in *DDJ* last month) are evidence that Forth is ending its formative years and is becoming an adult language. That being our view, we'll be leaving some of the residual wrangling to other, language-specific forums, and we'll concentrate on providing useful Forth tools.

The most important language in microcomputer software development today is C, and *DDJ* will continue in 1985 to be the best source for new programming tools in C.

Finally, *DDJ* will not lower its technical standards. Many readers have insisted that we not water down the technical level of the magazine. Heaven help us if that's bad advice, because we're going to follow it. On the other hand, don't be alarmed to see a page here and there given to humor, puzzles or programming competitions. Serious doesn't mean stoic.

Michael Swaine

Michael Swaine

Join the network

Develop a program for the new IBM PC Network.

Promote yourself into the top ranks of network programmers. Write new programs or modify existing ones to run on the local area network that's sure to be a best seller.

Capitalize on a chance to be in from the beginning with integrated business applications, productivity tools and office automation programs.

A high-level interface card makes programming for the IBM PC Network fast and easy. Full data sharing and byte locking capability, multiple servers, two megabits per second data transmission and a complete diagnostic package make the network equally as attractive to the end user.

For complete details about joining the network call 1-800-426-2700. Or write IBM Corporation, Editor, IBM Personal Computer Seminar Proceedings, 5Q9/4629, P.O. Box 1328 Boca Raton, FL 33432.

Little Tramp character licensed by Bubbles Inc., s.a.

IBM PC Network Specifications

IBM PC NETWORK ADAPTER

HARDWARE HIGHLIGHTS

MICROPROCESSORS
- 6MHz 80188
- 6MHz 82586

MEMORY
- 32K PROTOCOL ROM
- 16K RAM
- 8K NET BIOS ROM

VIDEO COMPATIBLE RF MODEM
- TRANSMIT 50.75 MHz (CH T14)
- RECEIVE 219 MHz (CH J)
- SUPPORTS 1000 NODES
- MAXIMUM DISTANCE 5KM RADIUS FROM HEADEND
- MULTIPLE SERVICES POSSIBLE

DIAGNOSTICS
- POWER-ON SELF-TEST
- ON-LINE MEDIA MONITORING

OTHER
- SUPPORTS DMA DATA TRANSFERS
- 2-MEGABIT/SECOND DATA RATE
- MID-SPLIT BROADBAND

FIRMWARE HIGHLIGHTS

OPEN ARCHITECTURE
- PEER-TO-PEER NETWORK
- OPERATING SYSTEM INDEPENDENT
- LOCALNET/PC™, PUBLISHED LAYERED PROTOCOL

FUNCTIONS
- BASE FUNCTIONS PROCESSED ON THE ADAPTER, NOT THE PC
- DISTRIBUTED NAME SUPPORT
- REMOTE PROGRAM LOAD
- 32 CONCURRENT TWO-WAY SESSIONS
- HIGH THROUGHPUT RATE AT SESSION LAYER
- CHARACTER SET INDEPENDENT

IBM PC NETWORK CABLING SYSTEM

IBM PC NETWORK TRANSLATOR UNIT

HARDWARE HIGHLIGHTS
- SINGLE RF CHANNEL CONVERSION
- ATTACHMENT OF UP TO 72 PCs WITH IBM CABLING
- ATTACHMENT OF UP TO 256 PCs WITH CUSTOM CABLING
- DATA ONLY
- ALLOWS NODES WITHIN A 1000-FOOT RADIUS

CABLE

TYPE
- STANDARD CATV MEDIA (75 OHM COAX)
- TREE TOPOLOGY
- CATV STANDARD F-CONNECTORS
- PREBALANCED BROADBAND NETWORK

KITS
- BASE EXPANDER (ALLOWS EXPANSION FROM TRANSLATOR)
- SHORT-DISTANCE KIT (1 FOOT ADDITIONAL CABLE)
- MEDIUM-DISTANCE KIT (400 FEET ADDITIONAL CABLE REQUIRED)
- LONG-DISTANCE KIT (800 FEET ADDITIONAL CABLE REQUIRED)
- CABLE AVAILABLE IN 4 LENGTHS: 25 FT., 50 FT., 100 FT., AND 200 FT.

MAXIMUM PCs AND DISTANCES SUPPORT	RADIUS FROM TRANSLATOR	PCS
• TRANSLATOR ONLY	200 FEET	8
• 8 SHORT-DISTANCE KITS	200 FEET	72
• 8 MEDIUM-DISTANCE KITS	600 FEET	72
• 8 LONG-DISTANCE KITS	1,000 FEET	72
• 8-KIT COMBINATION	200 to 1,000 FEET	72

IBM PC NETWORK SOFTWARE

DOS 3.1

EXPANDED SUPPORT FOR NETWORKING
- FILE SHARING
- RECORD LOCKING DOWN TO BYTE LOCKING

PROGRAM INTERFACE TO NETWORK SOFTWARE
- REDIRECTION CONTROL
- INSTALLATION CHECKING
- DIRECT EXECUTION OF NET BIOS FUNCTIONS
- MULTIPLE SERVERS

IBM PC NETWORK PROGRAM

FULL SCREEN INTERFACE AVAILABLE

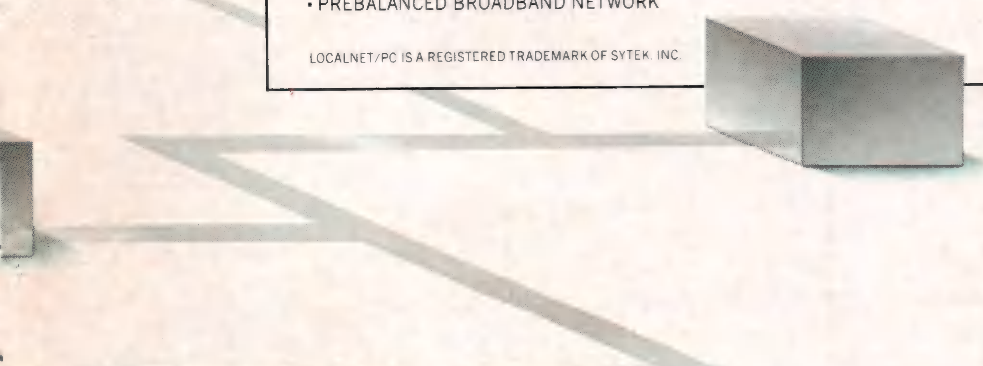
REDIRECTOR
- ALLOWS USE OF SHARED PRINTERS
- ALLOWS USE OF SHARED DISKS AND DIRECTORIES
- PROVIDES CAPABILITY TO SEND MESSAGES

FILE SERVERS
- SHARED USE OF NAMED DISKS OR SUBDIRECTORIES
- PASSWORD PROTECTION AGAINST UNAUTHORIZED ACCESS
- VARIETY OF ACCESS MODES SUPPORTED (E.G. READ ONLY)
- RECORD LOCKING TO CONTROL MULTIPLE UPDATES

PRINT SERVER
- SHARED USE OF NAMED PRINTERS
- PASSWORD PROTECTION AGAINST UNAUTHORIZED ACCESS
- AUTOMATIC SPOOLING AND QUEUING OF OUTPUT
- QUEUE MANAGEMENT FACILITIES ON SERVER STATION

MESSAGE SERVER
- INTERACTIVE MESSAGE EDITING/TRANSMISSION/RECEPTION
- PRESERVES FOREGROUND APPLICATION CONTEXT
- AUTOMATIC NOTIFICATION OR LOGGING TO DISK/PRINTER

LOCALNET/PC IS A REGISTERED TRADEMARK OF SYTEK, INC.



IBM®

Commodore Commentary

Dear *DDJ*:
The June 84 article on CP/M on the Commodore 64 is "just what the doctor ordered." I think that the Commodore will become a very important tool, but there is one fly in the ointment. The disk drive is s-l-o-w. A serial data link is not as stiff or noisy as flat cable (the dreaded RFI monster) but is inherently slower. Examination of the *Commodore 64 Programmers Reference Manual* seems to indicate something seriously wrong. It just couldn't be *that* slow. Even with 1000 picofarad loading, the 1000 ohm pull-up resistor would give a fast 1 microsecond time constant. Data handling in the CPU is the real problem, but even so, it should take less than 30 microseconds per bit. A 256-byte disk block should transfer in less than 62 milliseconds. It may be that cycle stealing by the display device requires huge operating margins. In this case, letting the CPU provide data clocking for both incoming and outgoing block transfers eliminates the need.

I may have missed something obvious, but that still does not excuse Commodore's "floppy tape" philosophy.

Yours:
Frank J. Wilson
2468 Elden St., Apt. J
Costa Mesa, CA 92627

More Chinese Forth

Dear Doctor:
As a longtime student of Chinese and a recent convert to Forth, I enjoyed Timothy Huang's article in the June 1984 issue of *DDJ* (No. 92). I agree that Forth should translate very well into Chinese. However, the use of the old phonetic symbols to spell out Chinese words is a giant step backwards. Overseas Chinese may know the old phonetics, but I believe that these symbols are little used on the mainland now.

It would be quite straightforward and much better to use China's own *Pinyin* for this purpose. This is not just another phonetic system; rather the Latin alphabet has actually been incorporated into the Chinese language. The letters are used for teaching and to provide a precise description of speech sounds.

For example, the Forth word OR could be typed in as HUOZHE; the Chinese will see this as the normal word for "or" (the regular tone marks *could* be used: HUÒZHĚ, but programmers could probably do without them and simplify the code). Likewise, ROT would be FANZHUAN, DROP becomes DIUDIAO, and DUP is CHONGFU. The byte-rich *Pinyin* versions could be shortened, e.g., like our ROT and DUP.

The main advantages would be that the words would be real Chinese words, and foreign computer consultants (who could already know *Pinyin*) would have easy access to the code.

Huang refers to the significance of the billion Chinese; most of them are on the mainland, however. Sooner or later, the overseas Chinese will have to come to terms with the big changes that have been made in the Chinese language during the last generation. In writing Forth, these changes may actually benefit the programmer.

Roger V. Swearingen
5333 Baltimore Dr., #158
La Mesa, CA 92041
(619) 697-3290

And the Shotgun Approach

Dear Doctor,
I must say that, as a new subscriber, it is a pleasure to have an issue arrive that demonstrates my own good judgment with regard to magazine dollars. While I'm not a C programmer, the C articles were a font of interesting algorithms and ideas. The core of my interest, though, was in the three articles directly addressing my interests—CP/M on the Commodore 64, Chinese Forth, and decision and cognition theory in Comments on Sixth Generation Computers.

With regard to CP/M on the C-64, I would add to the author's description of how to patch around having two 1541 disk drives the admonition that the 1541 was not really built for this job in the first place, and if you are going to have more than one disk drive on the Commodore-64, you should seriously consider getting a dual drive unit in the first place. The second comment is that, beyond the drive format availability, there is the fact that the 64 has a 40-column screen, and while you can get around this, it comes at a price that belies the 64's original value. Yes, there are a lot of Commodore 64s out there, but no, there are not a lot with CP/M; this situation is stuck in the old loop that it will take 64-formatted disks available in the real world to make CP/M a more popular option for the 64, and it will take more 64s with CP/M to make more 64-format disks available. The screen makes it all that much less likely.

Timothy Huang's article on Chinese Forth was very intriguing. It would be nice to see what matrix representations of the listed characters would look like and what resolution is necessary to produce them. I might also comment that in a subroutine-threaded Forth (in 6502), C: could simply be

```
: C: create –4 allot JUMP C; ' , ;
```

that is, back up over the 4-byte subroutine-threaded create routine, put the constant for the processor's jump instruction ($4C for the 6502) into the dictionary, and follow it with the cfa of the primary instruction. This demon-

strates the versatility of subroutine-threaded Forth—with direct executable machine instructions, you can jump directly into a definition. (For a discussion of subroutine-threaded Forth, see the June *Byte*.)

Ah yes, but the almost-gem of the issue was Michael Doherty's continuation of a dialogue on the future of computing. Almost everything Mr. Doherty said is towards the point, if not quite to the point. With a slightly stronger base in physiology, he could have come right out and said it—the human brain is not, repeat, not a Turing machine. We don't know exactly what it is yet, but we do know that the basic unit of the human brain is not only switched but also magnitude modulated, and that its output may act so as to switch other neurons, to modify the magnitude of their responses, or to do both simultaneously with the same or distinct target neurons. Push or pull it any way you prefer, and the result is still that the basic building block makes a difference. It may in fact *not* be possible to build a Turing machine with magnitude-modulated components; it is at any rate clear that this did not occur with the human brain. Here perhaps the useful metaphoric distinction is not that between quantity and quality but between number and quantity. You always have a distinct number of things. You never have a distinct quantity of something. When we improve our measurement capability to measure, say, an exact quart of liquid, we turn and look and, lo, what we have found is the number of atoms in a quart, and we have turned from measuring quantity to number. Please note that I am using these words in their squishy English senses and not in the precise Latin quantum of physics.

So much of Mr. Doherty's commentary was on the ball. He slips up on the discussion of information storage, however. He confuses information with the media on which it is stored. His oversight is not an uncommon one, after all. Who has seen information that isn't stored in some medium or other? Good question indeed, and the answer may just be everyone, for we don't know and are not yet even ready to address the question of where the quantum information of a quantum

particle *is* exactly.

The reason that this is to the point is that, if this quantum information is in a null space, then it would be, in actuality, medium-free information, because storage media don't fit in dimensionless spaces. Well, if quantum information can be there, it is possible to entertain the notion of keeping other information there, and then the storage "space" constraint on information is quite meaningless. That doesn't affect the main question of what will be processing the information, and it's not too hard to see that the obvious next step after putting discrete processors in parallel is to put them in parallel with distributed analog processors. This leaves open whether we can engineer better analog processors than the von Neuman machines running around putting out provocative magazines and the like.

Yours,
Bruce R. McFarling
600 W. Broadway #6
Granville, OH 43023

[*More letters on Sixth Generation dialogue next month. — Ed.*]

Turbo Tidbits

Dear *Dr. Dobb's*:
I enjoyed the review of Turbo Pascal in the June issue. I have been using Turbo Pascal for several months, and it is an excellent program, fully deserving the praise in the *DDJ* review. I thought it would be useful if I listed some of the problems I have discovered in the Turbo Pascal system. I am presently using version 2.0 of Turbo Pascal on an 8-bit (CP/M) system. As far as I have been able to determine, virtually all of the following persist in both versions 1 and 2.

1. The installation program reverses the definition of screen hiliting and unhiliting commands. There are several sources of confusion in the terminal installation program, which manifest themselves when nonstandard installation is attempted.

2. The listing program, TLIST, has several bugs, the most serious of which result in missing characters or lines and an improper number of lines per page, which puts the page break in the middle of the page for long programs.

3. In the 8-bit version, recursion is not permitted unless a special compiler directive is used. However, no error message appears either at compilation or run-time when code is recursive. This is a serious problem, because recursion may be indirect and hence not obvious (procedure A calls procedure B which calls procedure C which calls . . . procedure A).

4. When inline code is used, a variable identifier may be used, which, according to the manual, is replaced by two bytes which contain the memory address of the variable. In fact, the variable name is sometimes replaced by the address of the variable and sometimes by an address which contains the address of the variable (indirect addressing). Which of these is the case depends on whether the variable is global, local, a var parameter, or other parameter, and in some cases how the variable was used previously in the program block.

5. Turbo Pascal programs destroy all but about the first 30 characters in the command line buffer.

Even with these problems, Turbo Pascal is a great program and a bargain. However, since Borland was notified of some of these problems well in advance of the version 2.0 release, it is unfortunate that they didn't make any corrections in the new version or even include a list of known bugs in the documentation.

Sincerely yours,
Harry Demarest
Astro Research, Inc.
Box 74
Corvallis, OR 97339

Dear *DDJ*,
Regarding your review of Turbo Pascal in the June 84 issue, I agree that generally Turbo is a vast improvement over other compilers. There is one problem with Turbo which I find very disturbing, however, and that is the way it handles (?) integer overflow. The program (see figure below) illustrates the problem. A total program crash would be much more acceptable than to have this sort of viper lurking around. To Borland's credit, section 3.1 of their manual plainly states what will occur, and there is even an entry in the index, but still how hard could it be to fix this?

Donald G. Simpson
1121 Manchester Dr.
Raleigh, NC 27609

```
        Line 1 Col 1 Insert Indent B:INTBUG.PAS
program IntegerBug (input, output);
{ Demo to display results of integer overflow }
{ Turbo Pascal v2.0 MS-DOS configured for Zenith Z-100 }

var n: integer;

begin
        ClrScr;
        write ('Enter an integer between -32,767 and +32,767: —> ');
        readln (n);
        writeln;
        writeln ('The integer you have entered is: —> ', n);
        writeln;
        writeln ('Twice the integer you have entered is: —> ', 2 * n)
end.

Enter an integer between -32,767 and +32,767: —> 15000
The integer you have entered is: —> 15000
Twice the integer you have entered is: —> 30000
>
Enter an integer between -32,767 and +32,767: —> 32767
The integer you have entered is: —> 32767
Twice the integer you have entered is: —> -2
>
```

Figure

Dear Sir:

In the review of Turbo Pascal in your June issue, there is an apparent misunderstanding of the **with** statement. I refer to material in the righthand column of page 76. The following Pascal excerpts should clarify the matter.

```
     type
          rec2 =  record
                    s1 : integer;
                    s2 : boolean
                    end;
          rec1 =  record
                    r1 : integer;
                    r2 : rec1
                    end;
var
     a : rec1;
begin
     with a.r2 do
        s1 := 5;
     writeln(a.r2.s1)
end
```

This will obviously output the integer "5". Now amend the code part to read:

```
     begin
          with a do with r2 do begin
               s1 := 5;
               r1 := 4
          end;
          writeln(a.r2.s1,' ',a.r1)
     end
```

The output will be "5 4". Obviously the nested **with** statement qualifies with "a" where applicable and with "a.r2" where applicable. Finally, "with a,b,c . . ." is an abbreviation for "with a do with b do with c do . . .". So, the above fragment can be written:

```
     begin
          with a,r2 do begin
               s1 := 5;
               r1 := 4
          end
          writeln(a.r2.s1,' ',a.r1)
     end
```

I hope this sheds more light than it does confusion.

Yours very truly,
Herbert Kanner
211 Washington Ave.
Palo Alto, CA 94301

DDJ

DR. DOBB'S CLINIC

by D. E. Cortesi, Resident Intern

An Optimizing Technique

Early this year, your intern gave vent to the frustration he feels at the primitive designs of compilers for personal computers. All (or, at any rate, most) of them are simple one-pass, recursive-descent parsers. If you don't know what that means, study the Small-C compiler; it's a textbook example. In a recursive-descent compiler, the flow of control in the compiler echoes the syntactic structure of the source program (the syntax graph of the program at any point is described by the state of the compiler's dynamic call-stack when it has analyzed to that point). In a one-pass compiler, the object code for each text element is emitted as soon as that element has been parsed.

Such compilers are (relatively) easy to design, and they run quickly. However, they are incapable of doing any kind of global optimization—this operation requires the compiler to make multiple passes over the program. In a multipass compiler, the first pass (which can be designed as a recursive descent) validates the program's syntax, collects information in the symbol table, and probably builds a tokenized version of the expression text for later processing. Later passes can apply the information found in the first pass in such a way as to make the final object code smaller, faster, or both.

Several readers responded with long, thoughtful letters on the place of C and its compilers. David S. Tilton, for example, made the point that because C is a low-level language designed to give the programmer close control of the hardware, we shouldn't expect a C compiler to do much optimization. He said, first, that the language is so simple there aren't many opportunities for a compiler to do optimization, and second, that the language gives programmers the tools they need to do the optimization themselves.

Another respondent, Mike Meyer, made an architectural point. He pointed out that most data references in a recursive language such as C are to local variables and hence to the stack. "Unfortunately, the 8080 family just doesn't support such things well. What a local reference turns into is 'get the item so many words from the frame pointer.' On an 8080, this translates into several instructions just to load a local."

Well, these letters got us thinking. In 8-bit machines, as Meyer says, the code for accessing a variable with the dynamic storage class is much slower than that for accessing a variable with the static storage class. Local variables of a function are dynamic by default, but the programmer can give them the static storage class.

Now, every book on C programming advocates the use of lots of small functions—each with its handful of local variables—and these locals are just the pointers and counters that are most heavily used. But if you are using an 8-bit machine, all those locals will cost you heavily. "Everybody" knows that for best performance at least some of them should be declared static.

There are problems in doing so. You have to decide which functions are in the critical 20% and which of their locals are really important; you have to modify the source text and recompile. These are clerical tasks that the computer should do. Furthermore, if you make a local variable static in a recursive function, you'll cause a tricky bug. It isn't always easy to tell which functions can be called recursively and which cannot, especially if you are modifying someone else's program.

Finally, there is no speed penalty on dynamic locals in most 16-bit machines. In the 8086, for example, it costs no more time to access the stack frame than it does to access the data segment. Yet, if your 8-bit program is ported to a 16-bit machine, those static local declarations will port right along with it.

This seems like a perfect example with which to refute Tilton's point—a case in which the programmer could *not* do a decent job of optimizing and in which the compiler, even an 8-bit compiler, *could*. Shouldn't an 8-bit C compiler be able to tell which local variables should be static and which not? At the very least, the compiler could tell which functions contain no function calls at all; their locals certainly can be static. Furthermore, no two such functions can ever be active at the same time, so the locals of all such functions could be allocated in the same area of static storage.

Well, it's all very well to say a compiler *should* do these things, but is it really a practical thing to ask? We sat down one day and worked out the optimization scheme that follows. After describing the method at a high level, we'll consider its space and time requirements.

The Well-Behaved Function

The input to a C compiler is a source text that declares one or more functions. Each function is public; it can be called from other compilations.

The functions contain expressions, and an expression can include a reference to other functions. A function that refers to itself is directly recursive. A function can be indirectly recursive, in that it calls a function (that calls a function, and so on) that calls the first function again.

A reference to a function that is not declared in the same text is an external reference; the compiler can't know anything about the called function. In particular, a function that contains an external reference is potentially recursive, because the external function

might call (a function that calls, and so on) the first function again.

The functions we want to identify are those that are not recursive. Because the compiler's information is limited, we will take a conservative approach, assuming a function is recursive if there is the slightest possibility that it might ever be so. The functions we can identify as definitely nonrecursive we will call "well-behaved functions," or WBFs. A WBF can have its local variables allocated in static storage. We can give a recursive definition of a WBF:

- A function that calls no other function is well behaved.
- A function that calls only WBFs is well behaved.

The optimization scheme is based on this definition, but a lot of details have to be tended to first.

GLFs and TFs

The functions of the C standard library are named as external functions, yet they are almost always well behaved. We can think of only one case in which a library function is ill behaved. In some implementations, a file access function may, on encountering a disk error, make an indirect call to a user-written error function. In this case, the error function might conceivably call the function that called the library function in the first place, thus producing a recursion.

In the main, functions from the standard library are well behaved, and they are named in many user functions that, but for these references, would be seen as well behaved. Let us define a set GLF of good library functions that are always well behaved and use it to expand our definition of a WBF:

- A function that calls no other function is a Terminal Function (TF).
- A function that calls only GLFs is also a TF.
- A TF is a WBF.
- A function that calls only WBFs is a WBF.

We will detect WBFs in two stages: first, we will find all the TFs; then we will find the WBFs that are not TFs.

The Symbol Table

We are supposing a multipass compiler. The first pass performs the syntactic parse of the source text and, for every identifier named in it, builds a symbol table entry (STE). We are concerned only with the STEs built for function identifiers. Besides the other information the compiler needs, we will require these items in a function STE:

- glf: Boolean; true for a good library function
- idf: Boolean; true for an internally defined function (one defined in this source text)
- prf: Boolean; true for a possibly recursive function
- caf: Boolean; true when this function calls another function
- pwb: Boolean; true for a possibly well-behaved function
- twb: Boolean; temporary flag for

well-behaved functions

- wbf: Boolean; true for a well-behaved function
- asl: Boolean; the aggregate size of the local variables declared in this function
- osl: integer; the offset of static locals (when used)
- map: integer; used in mapping during pass 2

The use of these items will become clear as we proceed.

Before Pass 1

Before the first pass begins, the compiler must prepare a list of names of GLFs. The list can be coded into the compiler or it can be read from an external file. The latter course would allow users to extend the list with names of their own GLFs or to remove names if they should modify a library function in such a way as to make it ill behaved.

During Pass 1

During the first pass, the compiler must collect information that will be used to identify WBFs. These actions are in addition to the other things it does while parsing the source text.

When an STE is created for a function, the added variables listed above must be initialized. Except for glf, all the Booleans can be initialized to *false*. The name of the function must be looked up in the list of GLF names and the glf Boolean set *true* if it is found and *false* otherwise. The contents of the integer items can be undefined. When a function identifier is parsed in an expression or an initializer, an STE will be created as described (or merely found, if the function was declared previously).

When a function declaration is parsed, its STE will be created (or found, if the function is declared after it has been referenced). At that time, the idf Boolean must be left *false* if the declaration has the external attribute or set *true* if it does not. After a nonexternal function has been parsed, the asl integer must be set to the aggregate size of the local variables declared in the function.

When a function call is parsed, the compiler must record it as a tuple (*source, target*) in a list of such call tuples. In the tuple, *source* is the address of the STE of the current function, *target* that of the called function.

If the call is indirect through a pointer, the called function is unknown. In that case, the tuple should be recorded as (*source, source*). This makes the call appear to be a direct recursion. That's reasonable, because an indirect call could conceivably represent a recursion, either direct or indirect.

At the end of the first pass, the list of GLF names has served its purpose. Its space can be reclaimed.

Identify TFs

When pass 1 is over, an STE exists for every function identifier, and a call tuple has been recorded for every call. Now we can identify the terminal functions (TFs). To do so, scan the list of call tuples and, for every (*source, target*),

- if (*source* = = *target*), set *source*.prf = *true*
- else if (not (*target*.idf or *target*.glf)), set *source*.prt = *true*
- else if (not *target*.glf), set *source*.caf = *true*

The TFs are functions that call no functions or call only GLFs. They can now be marked as WBFs. Because they can't be recursive, their locals can be static. Because they call no user-defined functions, no two of them can ever be active simultaneously; therefore, their static locals can be allocated in a space that is common to all of them.

We will need two variables to handle storage allocation: slbot will contain the base offset of an area for static locals, and sltop will track the high-water mark of the area. Set both to zero. Then scan the symbol table and, for every function entry in which (idf and not (prf or caf)) is true:

- set wbf = *true*
- set osl = slbot
- set sltop = max(sltop,asl)

When code is generated for WBF, its locals will be allocated at an offset of osl in an area of static storage. The locals of a function not marked as a WBF will be given dynamic allocation as usual.

If it should prove that there are no TFs, nothing further can be done.

Map PWB Functions

We have established the TFs (which are also WBFs), but there are more WBFs to be found. They are a subset of the set of possibly well-behaved functions (PWBs). To locate this subset (actually multiple subsets), we need to establish a mapping from the PWBs to the integers $1..P$, where P is the count of PWBs.

Allocate space for an array of pointers to function STEs and call it F. Then scan the symbol table for PWBs. A PWB is a function STE for which (idf and not(prf)) is true. This includes the TFs plus all the functions that contain neither an indirect call, an external call, nor a direct recursion. (Some of these may yet be recursive. We will never identify them as such, but we won't call them WBFs, either.) Initialize P to zero. For each function STE,

- set pwb = idf and not(prf)
- if not(pwb), then iterate, else
- increment P
- set map = P
- set $F[P]$ to the address of the STE

Now the functions $F[1..P]$ are the (possibly) well-behaved ones, and the array F serves to map them to the integers $1..P$. If there are none, of course, nothing further can be done.

Cross-Map Function Calls

We must prepare an array of Booleans B having P rows and columns. Each row $B[r,*]$ will represent calling function $F[r]$; each column $B[*,c]$ will represent called function $F[c]$. Allocate the array and clear its elements to *false*.

Fill in the array with a scan of the call tuples. For each tuple (*source, target*), if both *source* and *target* are PWBs, set B[*source*.map,*target*.map] to *true*.

After this is done, the list of call tuples has served its purpose; its space can be reclaimed.

Discover WBFs

We will now make multiple passes over the array B. On the first pass we will find those functions that call only TFs. Because they do, they are WBFs; hence, their locals can be static. Because they don't call each other, no two of them can be active at once; hence, their locals can be allocated in a pool that is common to them (but distinct from the space used by TFs). In each later pass we will find a set of functions that call only WBFs found in prior passes. The same comments apply to each set of functions.

To be sure that the functions found in each pass don't call each other, we will use the twb flag as a temporary note of a function's well-behavedness until the pass is complete. We will also count the number of WBFs we discover. When we make a pass that discovers none, we will stop. The logic of the outer loop is:

- set discoveries = 0
- set slbot = sltop
- for *r* from 1 to *P*, perform inner procedure
- if discoveries = 0, break
- for *r* from 1 to *P*, if *F*[*r*].twb, mark *F*[*r*] a WBF

To mark a function as a WBF, we do these things:

- set twb = *false*
- set wfb = *true*
- set osl = slbot
- set sltop = max(sltop, slbot+asl)

The inner procedure inspects one function to see if it is well behaved. It needs to be performed only for functions that are not yet known to be WBFs. Its purpose is to set the *F*[*r*].twb flag.

- if (*F*[*r*].wbf), return
- set *F*[*r*].twb = *true* and perform inner loop

The inner loop runs a variable *c* from 1 to *P*. For every *B*[*r,c*] that is true, it checks *F*[*c*].wbf; if it is false, then *F*[*r*].twb must be set *false* and the inner loop can end.

At the completion of this operation, the WBFs have been identified in non-conflicting groups. Each group has been assigned an offset for its static locals, and sltop contains the aggregate size of all static locals.

Performance

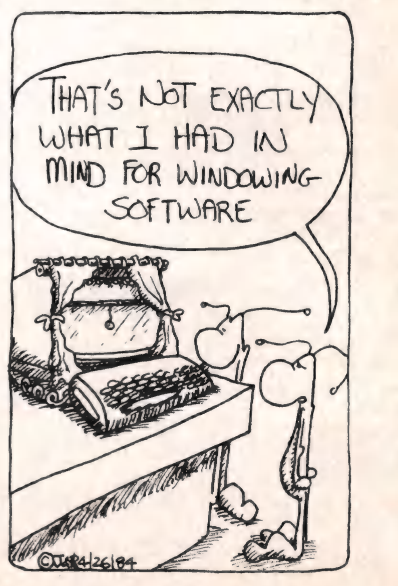
Through the point of identifying the TFs, this method shouldn't impose much of a load on a compiler, even an 8-bit compiler. The code added to pass 1 and the code to find the TFs are fairly small. The primary cost is in three blocks of space: the list of GLFs, the list of call tuples, and the extra items in a STE.

If space is a problem, the list of names of GLFs need not be complete. The list might contain only the names of the most common library functions. The more names in it, the more TFs that can be found. However, the only effect of omitting some GLFs is to make PRFs of some functions that could have been TFs; these will have dynamic locals when they could have had static ones, but the object program will still run.

The list of call tuples can also be truncated. It could be allocated in a small area of fixed size. Each tuple needs only 4 bytes, so an area as small as 256 bytes could contain up to 64 tuples—a useful number for small programs. When a function call is parsed and the tuple list is found to be full, pass 1 would mark the source function as a PRF and not record a tuple. As a result, a function whose calls were not completely recorded as tuples could never be identified as a TF or a PWB.

The extra STE information could impose the largest cost, because it adds several bytes to every symbol-table entry. However, that need not be true. Not every STE describes a function. The information unique to a function could be allocated separately and dynamically when a function is entered in the symbol table. The basic STE format would only need to allow room for a pointer to the function information, not the information itself. Furthermore, some of the STE items used here can be overlapped. The map integer may be a union with the osl integer, and the pwb and caf flags can be a union.

The cost of the second phase, locating other WBFs, is harder to analyze. It requires *F*, a vector of addresses of length *P*, but *P* is not likely to exceed 100, so *F* is unlikely to occupy more than 256 bytes. Furthermore, *F* could be built in the space previously occu-



pied by the list of GLFs.

The array B has $P*P$ elements, so it could become quite large if it used one byte per element—10K bytes when $P=100$. If it were stored as one bit per element, however, its size would be reasonable. In any event, the arrays F and B are transient; they can be discarded as soon as this part of the compile is finished.

The time cost of the second phase is potentially large. Each pass through the outer loop takes time proportional to P; there might be as many as P passes made. The inner loop takes time proportional to P, so at first blush the running time appears to be proportional to P cubed. It is not quite that bad, however, because the inner loop is only applied to functions that are not yet marked as WBF. Initially all the TFs are so marked, and more functions are marked on each pass. Furthermore, the inner loop can break off as soon as it is known that the current function calls a function that is not known as a WBF.

In fact, the inner loop need not be a loop at all, if B is stored one bit per element. The wbf flags of the functions $F[1..P]$ could be collected into a bit vector before the outer loop commences—while building F, for example. Then the inner loop could be reduced to anding the current row of B with that bit vector and comparing the result to the current row of B. This effectively removes the loop from the inner procedure.

That done, the time cost of the second phase becomes proportional to $2P$ times the number of passes made. The number of passes will be roughly proportional to the depth of nesting of function calls in the compilation. At one extreme, the compilation might define a chain of functions, A calling B calling . . . calling Z, a TF. Then one WBF would be found on each pass, and P passes would be made. At the other extreme, the compilation might define a set of independent TFs (and PRFs, but those are irrelevant), and only one pass would be made that would find no more WBFs.

Summary

In summary, we've outlined a method by which a C compiler could locate sets of functions whose local variables can be static and allocated in common pools. The time and space costs of the method are small for finding the first set (the TFs) and probably reasonable for finding many sets.

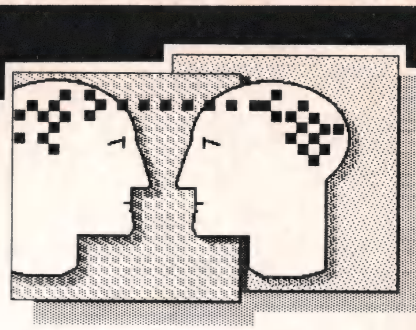
The method is described in the context of C, but it ought to be applicable to Pascal or Modula-2. Furthermore, in languages where every function is not automatically public, the TFs and WBFs have other useful properties. TFs are potential candidates for in-line expansion as macros, for example, and all WBFs are candidates for use of a short-circuit protocol for procedure calls that doesn't require a full stack frame.

We'd be delighted to hear from any designer who actually tries the method out or from any reader who finds a flaw in it.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 190.**

CP/M EXCHANGE



by Robert Blum

Everybody apparently has at least one true-life story to relate when the topic of conversation turns to humiliation. I'm no exception. I used to have several favorite stories; I now have another.

Several months ago I decided that I soon would have to replace some of my then dreadfully old S-100 hardware because I was beginning to experience excessive down time. Except for the cost of an upgrade, I wasn't at all upset at the prospect of buying some new hardware. I had been waiting quite a while for a sound reason to move ahead and bring up a higher performance system.

Many new high performance boards had become available since my original purchases. This caused me great consternation when deciding what I should buy. I wasn't unhappy with the performance or features of the boards I had at the time; they were simply old and needed replacement. As it finally worked out, I didn't find anything that better served my needs.

I elected first to renew my 4 MHz CompuPro Z80 CPU board with a more recent revision level of the same model, capable of 6 MHz operation. I knew my present memory cards would function perfectly at the increased clock rate because they were rated at 12 MHz, as were the I/O boards I was using.

I was not nearly as confident about the floppy disk controller that I had purchased from a different manufacturer. Of all the boards in my system at that time, the floppy disk controller was the most complex and the one that I knew the least about. I wasn't sure that I would be able to find a hardware solution to a speed-related problem if one were to occur. However, if at all possible, I wanted to keep the board because of the time I had spent fine-tuning my BIOS code. The thought of reinventing the wheel for another disk controller board didn't excite me in the least. Fortunately, a phone call to the manufacturer brought good news: I could upgrade to the latest revision level board, which was guaranteed to be 100% software compatible with the old one and to run flawlessly at 6 MHz, all for only a nominal upgrade charge.

While waiting for the new disk controller to arrive, I ran diagnostics constantly for several days. During that time not one error occurred that I could attribute to the new hardware.

Soon the new disk controller arrived. I wasted no time getting it ready for use and couldn't have been happier when it booted for the first time from drive A:. All seemed rosy until several programs on the A: drive turned up lost. Listing the directory showed that the programs were definitely missing; in their place were several meaningless filenames. I began to feel a bit uneasy because the last time I saw a display of this type was when I helped a friend rebuild his disk directory after a mysterious system crash.

I executed my disk utility program, which allows sectors to be displayed in hexadecimal, to see what was causing these unknown filenames to appear. I started my search with data group zero, which on all standard CP/M systems is the start of the directory area. About halfway through data group one, I found a sector that had been completely zeroed. I didn't have a hint as to what could have done this but suspected that the disk controller might be more clock rate sensitive than I had been led to believe. I slowed the CPU down to 3 MHz and inserted another disk into drive A:. Again the system booted on the first try, but this time it ran without destroying the directory.

This verified my first suspicion that the disk controller was speed sensitive. However, the manufacturer had told me that running at a faster rate would be no problem. I called again to find out if I had done something wrong or possibly had not configured the controller board correctly. I was told to check several wiring changes and was again assured that all would work as planned after the bugs were shaken out. I dutifully went about checking for any mistakes I had made during setup—without result.

Having thoroughly checked the hardware, I turned to my BIOS. The only area I could find that appeared in the least suspicious was a section of code in the warm boot routine that flushed any active deblocking buffers to disk when a program ended. I thought there was an outside chance that the disk heads were being moved before the deblocking buffers were written. I couldn't imagine this happening but commented out several instructions for testing purposes. After installing the modified version of the BIOS on the system tracks of a test disk, I again speeded up the CPU to 6 MHz and booted from disk. Once again, after several disk accesses, a sector in the directory was destroyed.

At this point I had tried everything I could think of. Rather than continue to shoot in the dark, I turned my troubled machine over to a hardware magician for repair. Several days later my system returned running flawlessly at 6 MHz. I was told that the solution was a few minor hardware adjustments.

For months now I have been running the system without any further hardware problems. However, a few software situations have cropped up. I use a fairly well-known data base package for most of my application-oriented tasks. After installing the latest release of this software, I suddenly began losing large portions of my indexed data files. Investigation showed that the data content of all the records was present in the body of the file but the indexes were being corrupted. Several phone conversations with the technical support people

proved fruitless, so I wrote a couple of test jobs that I knew would consistently fail on my system and sent them along for analysis with my distribution disks and a problem report.

A short while later the support people returned my package with a note saying that my test jobs had failed on their machine as well. They further acknowledged that a problem exists but they were unable to pinpoint the source since it appeared to be a CP/M idiosyncracy. In the meantime I could apply a temporary fix, which involved adding several instructions to each of my jobs to ensure that the data buffers were saved to disk after each update.

A practically identical situation reared its head a couple of weeks ago. I was in the process of installing a new release of my favorite BASIC interpreter. After I had completed the configuration process and was trying to execute the new interpreter, a message appeared on my CRT instructing me to run the configuration process before using the BASIC runtime system. That sounded like a reasonable request—except I had already completed that step.

A call to the software manufacturer proved to be a valuable lesson for me. He informed me almost right from the beginning of our conversation that my problem was due to a faulty BIOS. He explained that his configuration utility reads the first sector of the program and modifies it according to answers given by the operator. At the conclusion of this interaction the changed sector is rewritten to disk and the file is closed. Unfortunately, CP/M 2.2 does not automatically flush deblocking buffers when a file is closed. Under normal circumstances this procedure works fine; problems occur if the BIOS warm start routine does not flush the remaining deblocking buffers when a program ends. This wasn't exactly what I wanted to hear, and I held my ground by refusing to give any credence to a possible bug in my BIOS.

The fellow with whom I was talking wasn't about to be swayed from his convictions, which made me think twice about what he had to say. It didn't occur to me until the next day that months ago I had commented out several instructions in the warm start routine of my BIOS while trying to find a disk controller problem and in my haste had not returned them to their original order. After I did so, both of my software packages began to run properly.

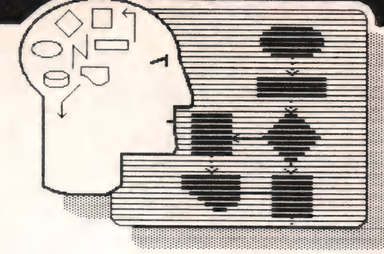
If you have experienced problems of this type, you might want to look through the warm start portion of your BIOS listing for a sequence of instructions similar to the following:

```
LD    A,HSTWRT      ;IS HOST BUFF
                    ;ER ACTIVE
OR    A             ;*
CALL  NZ,WRITEHST   ;YES—WRITE IT
                    ;TO DISK
```

Depending on whether your BIOS was written from scratch or patterned after the DRI examples, the instructions in the listing may not be even close to those in your BIOS. Nonetheless, what you are looking for is a section of logic that tests for and flushes any remaining active deblocking buffers. **DDJ**

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 191**.

THE SOFTWARE DESIGNER



by Michael Swaine

The following discussion never took place.

The software designers and critics whose words appear in this article have never gathered as a group to argue the design of software. If they had, it is doubtful that the precise exchange of views pieced together here would have resulted. But all the words I have quoted were actually spoken by the persons to whom they are attributed, and I have attempted to avoid changing the sense of any utterance in changing its context. I don't pretend that I have left every nuance intact.

The error of interpretation I am most likely to have committed is to make the speakers appear more dogmatic or polarized than they actually are. I have placed opposing views next to one another here to focus attention on the differing views; the undesired side effect of this is to make the contrasted views read more strongly than they would in isolation (or in their original context). Just remember that these people are all more complex individuals than their two-dimensional projections onto these pages.

The Participants

Rob Barnaby is the author of one of the best-selling software packages in history, WordStar. The compleat programmer, Barnaby delivered the original WordStar to MicroPro on disk, debugged and ready for duplication, along with camera-ready copy for the manual.

Lee Felsenstein is a hardware designer whose credits include the Pennywhistle modem, the Sol-20, and the Osborne 1.

Paul Heckel is the president of Quickview Systems, a software designer, and the author of *The Art of Friendly Software Design.*

Philippe Kahn is the president of Borland International, maker of Turbo Pascal.

Giacomo Marini is the vice president for software development for Logitech, maker of Modula-2 compilers.

Anthony Skjellum is president of Pyramid Systems, a software designer, and *DDJ* 's C and Unix columnist.

Pierluigi Zappacosta is the president of Logitech and formerly developed software on early 8-bit microcomputers.

It began last spring. Borland International's Turbo Pascal was beginning to attract attention, and editor Reynold Wiggins and I drove to Scotts Valley to talk with Borland's president, Philippe Kahn. In the course of our discussion, the following exchange took place:

***DDJ*:** Turbo Pascal is doing very well. Sidekick looks interesting. What's next? Will you do an implementation of C?

Kahn: C is a disease. When I see people writing spreadsheets in C, I think, "They're out of their minds." It was designed to write operating systems. Modula-2 is good for that [writing spreadsheets]. We'll do a C. We'll do a C because everyone wants a C. But in Europe C is seen as an American disease, and here people are trying to spread it.

Now, C is a programming language originally developed at Bell Labs for use in writing operating systems; it has recently become popular as a general-purpose programming language. But not, apparently, with Philippe Kahn. Struck by the vehemence of Kahn's reaction, I started talking with software designers and critics about what they thought the real differences were between C and Modula-2 from the perspective of software design, and the result was this pseudo-roundtable discussion.

Zappacosta: I'm surprised, in a way, that C was invented on the East Coast, because it would be the perfect language to have been invented on the West Coast.

Barnaby: I don't follow that. I think it's more a distinction of academic and scientific vs. free-flowing, open-ended design work.

Heckel: I think the point is not East Coast vs. West Coast, especially since there's so much movement between coasts, but rather the academic environment vs. the let's-get-the-job-done environment.

***DDJ*:** Bell Labs is not academic?

Heckel: OK, you might argue that Bell Labs is really an academic environment, but I point out that the charter of that lab was not to develop a language like C. C was a side effect. They decided that they wanted a simple operating system, so they developed an early version of Unix; they decided they wanted a high-level language, so they developed C. None of these things were what they were supposed to be doing. They were designing for themselves, focusing on tools for their own immediate needs rather than on an academic concern with how pretty an algorithm would look on paper. Modula-2, on the other hand, was a follow-on of Pascal that came out of the academic community. And Pascal was designed for pedagogical purposes, for the way the world ought to be.

***DDJ*:** I think Pierluigi was drawing on a more general West Coast stereotype, the free-wheeling, laid-back Californian. And whether or not the stereotype has any validity, I wonder if there isn't something to the idea that C is well designed—even if less deliberately designed than Modula—for a freer, less-regimented kind of programming. And if so, what does that mean?

Zappacosta: Yes. On the West Coast,

people say, I want to do whatever I want; I am my own boss. Why not? In the States you can change your last name, so why shouldn't you use C?

Skjellum: Well, freedom is an issue. But it can be a very practical matter. You sit down to write a program and you don't want to fool around with a lot of academic rules and constraints.

Heckel: Obviously the Bell Labs people tried to integrate as much of the knowledge they had gained regarding compiler construction and design as they could, but without getting in the programmer's way, still giving people the freedom they need. When you're designing real operating systems and real compilers, you try to give yourself freedom rather than a straitjacket.

***DDJ*:** It sounds like getting the job done is also an issue. Paul and Tony have both characterized C as a get-it-done language, and Paul has contrasted it to the structured, modular style of Pascal and Modula. But Lee, you've been talking to people recently about your own get-it-done approach to design, and what you've been saying sounds structured, modular.

Felsenstein: I'm getting into computer programming a bit more. First of all, I can't avoid it. No hardware designer today can ignore software. Secondly, for our project for the Community Memory terminal we have to make a bit of clever firmware. I intend to do that in one day, but I don't intend to do it myself. I'll spend about three weeks working out the structure of the system without getting down to the details. Those will then be filled in by a small army of coders—volunteers, in this case—who will have received specifications and will sit down for a few hours to code small routines.

***DDJ*:** You're approaching software design from the point of a hardware designer.

Felsenstein: It's similar to building at the level of the chip. You work from top down, build a structure, and at the last minute fill it in. To build software like hardware, I am finding, requires a fair amount of discipline to avoid rushing ahead, and it also requires that you be able to visualize the construct.

Heckel: I'm not against structured programming, but you don't hire a musician because he can read music better than the other guy; you hire him because he plays better than the other guy. The aesthetics of the algorithm is one of several factors one uses in evaluating software. The problem with the direction of structured programming is that there are more and more ways to do something wrong.

***DDJ*:** But structured programming, modular design, the approach that Lee just described and that Paul talks about in his book, that's all language independent, right? C has perfectly good tools for writing structured programs, and you can write modular code in any language. So how, specifically, is C any "freer" than other languages?

Skjellum: C doesn't force as many assumptions on the programmer. Think about the assumptions that are forced on you, in many languages, by the compiler. For example, you are forced to use intrinsic functions. In Fortran, all the I/O is intrinsic, and the compiler makes assumptions to help that along. And there are some traps that this leads you into. Think of the Write statement; the expression within the parentheses can be quite complex, and you might expect it to be handled by the compiler, but the compiler just passes it along to the runtime package. In C, you would know better than to make that mistake; the compiler doesn't play favorites.

***DDJ*:** In C, the compiler has no prejudice?

Skjellum: No; it neither helps nor hurts. In Fortran, Write statements are helped out by the compiler. Pascal is like that; it makes certain assumptions. Think about the Writeln command: you give it an integer argument and an object to print out. That rigidity is frustrating when you're trying to get a job done. What I want is to be able to define, say, my own I/O functions at a level where they look like the built-in elements of the language. In C, I can.

***DDJ*:** Modula-2 is being touted as the language that Pascal should have been, but aside from its roots in Pascal and an impression that it takes some freedom away from the programmer, I think most of us are a little vague about what Modula does and doesn't do. Does it, for example, force the kinds of assumptions that Tony wants to avoid?

Marini: In our version of Modula, we include the source code for some things that can be customized. Pointer management, the loading of overlays, how you manage interrupts: you may want to customize these things. Also, we distribute the sources for three or four library modules that are in some way environmentally dependent: the keyboard module, the display module, the RS-232 module. You can take the supplied modules as examples for designing your own. The flexibility is limited, but it's there. But the thing that makes Modula the language of the 80s is that you have separate but not independent compilation. You have a check at compile time between modules, so if you import a procedure from another module, you have to write the definition for the exporting module. When you link, you are sure that all the modules are version consistent. You have strict consistency checks between modules.

***DDJ*:** You must admit, that sounds constraining.

Marini: If you are the only programmer around, it can be a little annoying. The Modula approach certainly uses strong discipline. But as soon as you have three programmers working on a project, this is really invaluable. Modula can handle complex programming tasks. I think that this is critical. The only other language that does this sort of thing is Ada. For people who understand and like the advantages of compile-time checks and strong typing, Modula has the chief advantages of Ada without the complexity of Ada or the unavailability of reasonable Ada compilers.

***DDJ*:** Can you quantify the distinction between simple and complex?

Marini: I think the big difference is between the one-programmer and the multi-programmer project. That is the first step. The second step probably happens at about four or five programmers. From there up to a thousand programmers probably doesn't change very much. The big step occurs when you go above four or five programmers. Then you get into the complexity of communication within a team of software developers that cannot be based on simply informal, verbal communication. You start needing, absolutely needing, written specifications. When you have four or five, one will leave within the year. The probability is that they will not all have the same

style, the same feelings, the same taste, the same choices. You need some sort of interface management. You may want a network, a common library, other little things that are not so little in the end because they consume a lot of effort.

Zappacosta: And the DoD certainly knows quite a bit about that, because they specified Ada.

Skjellum: Ada exists for people who are writing a program that has to run for 20 or 30 years and they have to be sure nothing will go wrong. They're so worried that something might go wrong in their program that they have to force excessive rigidity in the design modules. So they invented Ada. Ada has very specific rules and a tremendous instruction set. I think of it as the Language of Ten Thousand Features.

Zappacosta: In some sense you can say they went overboard; yes, maybe. But when you deal with the complexity of programming that they deal with, maybe you understand why they went overboard. I think that somehow Modula took the right ideas from the right places. It's not so complicated that you have to have the DoD pushing you to use it. But on the other hand, it's appropriate for complex development like the people who are reprogramming air traffic control software throughout the country. If you propose to these people to use C, they cannot. They have I don't know how many people working on the project, and they are not sure that things would work together. So C is not an alternative.

Skjellum: I don't know much about Modula. And I'd like to keep it that way. But I must say that there are some nice features of Ada: more flexibility in the defining of data types, for example.

***DDJ*:** It seems that we all agree that C allows the programmer the freedom to do things in unorthodox ways, that Modula and Ada force the programmer into orthodoxy, and that the difference of opinion is over whether and when the loss of freedom is justified.

Zappacosta: The constraints of a language like Ada or Modula gain you something in functionality, but yes, the incremental gain costs something. There are two ways to go. One is toward simplicity—the pfs approach—everything is simple, manageable. With the other you get Ovation, Framework, more complicated pieces of software. Unfortunately, for certain applications you haven't a choice. If you want to control all the airplanes in the United States' sky . . . .

***DDJ*:** You can't use the pfs approach?

Zappacosta: Probably not. Logitech is positioned in a strange place. We offer Modula-2 for the microcomputer, and in the microcomputer you have the choice you don't have on larger machines: simple or complicated. Certainly we are not going after the simple applications. The fact that after the 8086 version we came out with the VAX version shows that what we have in mind are serious software developers.

***DDJ*:** Are you saying that you have to have a VAX to be a serious software developer?

Zappacosta: Well, even for applications that eventually go on a PC, development on a mainframe is becoming more common. That's why it's harder to start a software company. You have to have a VAX; if not a VAX, you have to have a 68000-based machine. When I started programming micros in 1977, we had a CP/M machine that was the target machine and the development machine. Back then, the software couldn't be too complicated; the hardware was most important. Now things are changing.

***DDJ*:** Software is getting more complex, requiring team programming, which in turn requires the overhead management that languages like Modula, with consistency checking and strong typing and so on, offer?

Zappacosta: Yes, but besides the number of programmers, the other parameter to measure complexity of software is the need for this software to change and evolve. If you write software and never change it, if it is a great hack and nobody can put his hands in there, while it works you can accept that level of functionality and reliability and leave it there. But there are many applications that are alive. And then it's different. If you have to put your hands into a system continuously, then you have this problem of maintenance. You have to understand the code of another guy, you have to port the system to a new machine, and that's another discriminating factor.

Skjellum: But you can do that with C. You can write machine-specific soft-

ware, but you can also write software that is portable.

Heckel: Yes, do you know the concept of *lint* in C? It's referred to in the C manual. The *lint* program takes a C program and it finds all the things that Pascal tends to find and it gives you a list of them. Then if you really want a program that doesn't have any of those kinds of things in it, you can fix the C program.

***DDJ*:** There really are two philosophies here, aren't there?

Zappacosta: Well, you know, every time we present Modula we go the easy way. We say, if you plan to use Pascal, Modula is a better alternative. I think there is little discussion there: Modula is indeed the better language. But C . . . . Of course there is the large C market that we would like to tap, but we are careful. We are not eager to start a religious war. There are two different kinds of people who defend C. One says, I don't need another language. That's either because their requirements are not that tough and they really don't need another language, or because of ignorance. When Bill Gates told Bobo [Daniel Borel] that he didn't need another language, I think that that was ignorance. After all, considering that he made his living writing BASIC, it's not surprising that he sees already too much for what he needs. But he has a large company, and now even Microsoft is getting behind schedule and not completing things. Maybe it is ignorance and he is making a mistake not to consider alternatives. But other people like C as a matter of personal taste; it's an expression of personal freedom. This second group we take great care not to touch.

Skjellum: At least compared to Pascal, C is a more general-purpose language; that's why I like it.

***DDJ*:** Even granting, for the sake of argument, that team programming is the wave of the future, still there are costs associated with every decision. What do you lose in moving from the approach of the lone programmer who codes down on the bare metal to the more rigid, Modula or Ada team programming approach?

Barnaby: It seems like to do that you've got to pretty well spec out what you're doing first. So you can divide it into pieces and design interfaces among the pieces. Which means you have to understand what you're doing first. And that isn't how you create products that are leaps in the state of the art. You get a good idea, you divide it up very roughly into pieces, and later find that they weren't the right pieces and you redefine. I find that I have to be freewheeling. That's my experience in doing programming. WordStar grew out of a video program, and as I worked on it I kept coming up with a better way to do it.

***DDJ*:** What one hears repeatedly is that many of the breakthrough software products are developed by programmers for themselves, not as products. Is that just folklore?

Barnaby: No. Look at Unix and C. I'm trying to get back to that approach after not having produced anything significant since WordStar and after having noticed that I wasn't approaching software designing in that way. I'm currently working with some very vague ideas that may lead to developing some new techniques to do certain things. It's creative exploration. It might or might not be incorporated into a product.

***DDJ*:** So, Rob, you're going back to an earlier, freer approach to design. And clearly an individual approach. But Pierluigi and Giacomo, you seem to be saying that the times have outgrown that approach.

Marini: Three examples: VisiCorp, Microsoft with Windows,Digital Research with Concurrent DOS. In all three cases we have consistent out-of-schedule behavior. These were basically all companies that went from very small development groups to significantly larger development groups, and probably without project management.

Zappacosta: The fact that the microcomputer industry has been so innovative, so creative, probably comes from the fact that the past was forgotten. The industry was new; the machines were simple and small. And simple tools and smart people produced good results. They proved that there is also a side to programming that is not represented by the Mythical Man-month. But then the industry grew, the machine grew in power, and now to reach the vast majority of people you have to use a different interface. The traditional solutions used in the microcomputer industry are no longer adequate. In the Macintosh you have 64K of highly optimized, hand-coded software just to move the mouse. Unfortunately this trend may make creativity more difficult to foster.

Marini: Well, the market is getting more conservative. Interface compatibility is becoming a little like the story of the awful IBM keyboard.

***DDJ*:** The awful IBM PC keyboard or the awful PCjr keyboard? IBM is backpedaling fast on the PCjr.

Marini: Maybe this is still reversible, but certainly the user interfaces to things like Lotus 1–2–3 are not easily reversible. The point is that the market builds up inertia.

***DDJ*:** Lotus must be hoping that users can unlearn the 1–2–3 interface, or Symphony won't sell. But I guess you're saying that users now have expectations, and those expectations constrain what software designers can do with their creativity?

Marini: Yes. I suspect that the creativity is still there. Americans are so conservative. In the U.S. you can't change the color of the dollar. In Italy, we change the color of the money every three years.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 192**.

More dBASE II Programming Techniques

by Gene Head

In my last article (*DDJ*, June 1984) I described a machine language subroutine called ZIP-CHK.ASM that partially validated zip codes and could be used by dBASE II using the LOAD FILE, SET CALL TO ADDRESS, and CALL VARIABLE commands. I now realize that older versions of dBASE II do not support the LOAD FILE command that facilitates the loading of machine language modules from assembled HEX files. However, most early versions of dBASE II (pre v.2.4) *will* support the SET CALL TO ADDRESS and CALL VARIABLE, even though these commands may be undocumented in the user's manual.

In this article I will explain these and other undocumented features of early versions of dBASE II. I have access only to versions 2.3 and 2.4, so you should check other versions thoroughly to determine whether the feature actually works as I will describe it. I would appreciate hearing about any results you may have on versions prior to v.2.3; I will catalog these findings and pass them on in future articles.

In addition to the undocumented features, I have included a description of the HEX file format and a dBASE II command file that will simulate the LOAD FILE command if you can get the undocumented PEEK and POKE to work. After reviewing the format of the HEX file, you should be able to write a LOAD simulator in any high level langauge using LOAD-HEX.CMD (listing, page 27) as a guideline.

Gene Head, 2860 NW Skyline Dr., Corvallis, OR 97330.

"Most early versions of dBASE II will support the SET CALL TO ADDRESS and CALL VARIABLE. I will explain these and some other undocumented features of dBASE II."

Pre-v.2.4 Undocumented Commands

These undocumented features appear in early versions of dBASE II. PEEK (aaaaa) will return the contents of a decimal address; POKE aaaaa,nnn will fill a decimal address with a decimal value; SET CALL TO aaaaa will set the call address to a decimal value; and CALL VARIABLE will call the address as defined in the SET CALL TO command and pass the string VARIABLE, as described below.

Upon entry of the subroutine, the HL register pair points to the location of the passed variable. The variable is stored in the format LENGTH BYTE followed by the actual string characters. For example, if the variable ZIP represents the string 97330, then upon CALLing a subroutine HL will point to a series of locations with the following data bytes in hex: 05, 39, 37, 33, 33, 30. The HL pair points to the 05, indicating there are five bytes in this string; 39, 37, 33, 33, 30 are the hex bytes that represent the ASCII string 97330. You can modify the string, but you must *not* increase the length of the variable. If the string becomes shorter, you should fill it with trailing blanks.

TEST(variable) will test the variable and return the following: −6 if the value is NUMERIC type, 0 if the variable does not exist, 1 if the variable is LOGICAL type, and nn (a number be-

tween 1 and 255) to indicate the length of a STRING type variable.

Loading a HEX File

The HEX file is simply an ASCII text file that holds information about what bytes go where. The LOAD.COM program usually is used with the HEX file to create an executable COM file. The format of any HEX file is shown in the figure at right.

```
:10A470002356235E22F0A401080021F2A47EFE07E9    <— line 1
:10A48000CACFA4BACA8BA409C37DA4237EBBCA9633    <— line 2
            (30 lines were deleted in this example)
:10A67200575632343732363857493533303534391 4   <— line 33
:09A682005759383230383331 00E9                 <— line 34
:0000000000                                    <— line 35
```

Figure

In a HEX file every line ends with a carriage return followed by a line feed, and every line begins with a colon. In line 1 (see figure) the two bytes following the colon (10) are two ASCII characters that represent the HEX value in the range 00 – FF (0 – 255 decimal). I will call this value the NUMBER OF BYTE-PAIRS value. We will get back to it later.

The next four bytes (A470) are ASCII characters that represent a HEX value in the range of 0000 – FFFF (00000 – 65,535 decimal). I will call this value the FIRST LOAD ADDRESS.

The next two bytes (00) are ASCII characters that are reserved. That means I don't know what they are for. However, in every HEX file I have ever looked at these two bytes are always 00, and for this article I don't think it is necessary to define them exactly. The same can be said for the last two bytes on each line (E9 in line 1). A checksum perhaps?

The intervening bytes (02, 35, 62, 35, etc.) should be viewed as byte-pairs. Remember the NUMBER OF BYTE-PAIRS mentioned earlier? Well, that is exactly the number of these byte-pairs we will load into consecutive memory starting at the FIRST LOAD ADDRESS. Each of these byte-pairs are ASCII characters that represent a HEX value in the range of 00 – FF (0 – 255 decimal). Given the example of line 1, the address locations should be filled with the corresponding byte values as follows:

A470	23	A478	08
A471	56	A479	00
A472	23	A47A	21
A473	5E	A47B	F2
A474	22	A47C	A4
A475	F0	A47D	7E
A476	A4	A47E	FE
A477	01	A47F	07

If your high-level language does not support a LOAD function, you can simulate one as long as you can read a textfile and PEEK at and POKE into memory. The command file LOADHEX.CMD (see the listing) was written for dBASE II. dBASE II only understands decimal numbers, and most of the code is for converting two ASCII bytes to a decimal number to be POKEd into memory. Understanding the format of the HEX file should enable you to write your own HEX loader simulator in any language.

Using LOAD-HEX.CMD as a guide, process line 1 of the HEX file. Extract the load address (position 4-7 in the string), the number of byte-pairs to POKE (position 2-3), and the individual byte-pairs (beginning at position 10). POKE each byte-pair then process the next line of the HEX file. Continue processing sequential lines until the NUMBER OF BYTE-PAIRS is equal to zero. The HEX file is now loaded into memory and ready to CALL as a machine language subroutine.

DDJ

dBASE II Listing

```
*        LOAD-HEX.CMD
*
*        File to simulate the LOAD feature of dBASE II 2.4 in
*        earlier versions that do not support this feature.
*
*        Verify that PEEK and POKE commands work before typing
*        in this file and watch the syntax especially '(' & ')'.
*
*        From:    Head Quarters
*                 2860 NW Skyline Drive
*                 Corvallis, Oregon  97330
*                 (503)  758-0279
*
*        Copyright 1984 by Gene Head - All rights reserved
*        For private, non-commercial use only.
*
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

(Continued on next page)

 (Listing Continued, text begins on page 26)

```
SET TALK OFF

* --->  First time initialization
*
* The following need be done only once
*
* First CREATE a file named HEXDATA.DBF with ONE
* field NAMED DATA of CHARACTER type, forty-four
* (44)  characters in length.
*
* APPEND FROM the SUBROUTINE HEX file using the SDF option.
* If you were going to use the Subroutine ZIP-CHK.HEX file then:
*
*        .
*        .USE HEXLOAD
*        .APPEND FROM ZIP-CHK.HEX SDF
*        .
*
* <---  End of first time initialization
*  Each line of HEX data will be in the field named DATA
USE HEXDATA

*   POSITION IN THIS STRING IS THE DECIMAL VALUE OF THE HEX CHARACTER
*   (SEARCHING FOR A '0' RETURNS 0)
STORE '123456789ABCDEF' TO HEX

* --->  COMPUTE THE BASE LOCATION OF THIS LOAD (THIS FUNCTION IS OPTIONAL)
STORE STR(((@($(DATA,4,1),HEX)*16 + @($(DATA,5,1),HEX)) * 256) +;
  (@($(DATA,6,1),HEX)*16 + @($(DATA,7,1),HEX)),5) TO CALL:ADR
SET CALL TO &CALL:ADR
* <---  END OF LOCATION OPTION

DO WHILE $(DATA,2,2) <> '00'
  * Compute how many bytes to POKE from this line of the HEX file
  STORE (@($(DATA,2,1),HEX)*16 + @($(DATA,3,1),HEX)) TO COUNT

      *  Get the starting POKE address
      STORE ((@($(DATA,4,1),HEX)*16 + @($(DATA,5,1),HEX)) * 256);
       + (@($(DATA,6,1),HEX)*16 + @($(DATA,7,1),HEX)) TO ADDRESS

      * We POKE the last BYTE-PAIR on the line of the HEX file
      * and work our way back to the first BYTE-PAIR on the line.
      DO WHILE COUNT
        STORE STR(ADDRESS+COUNT-1,5)+','+STR(@($(DATA,COUNT*2+8,1),;
          HEX)*16+@($(DATA,COUNT*2+9,1),HEX),3) TO BYTE
        POKE &BYTE
        STORE COUNT -1 TO COUNT
      ENDDO WHILE COUNT

  * GET NEXT LINE OF HEX DATA
  SKIP

ENDDO WHILE $(DATA,2,2) <> '00'
USE
```

End Listing

Simple Calculations With Complex Numbers

by David D. Clark

This article will introduce you to the complex numbers and some of the methods used to perform calculations with them. Despite the title, the ideas involved are quite simple. First, however, a little background on the rational, irrational, and real numbers.

Most people are familiar with the rational numbers, whether they know it or not. Rational numbers are the numbers we use in our everyday lives, like the 25 cents we pay for a newspaper or the 1/2 teaspoon of vanilla extract required for a certain recipe. Rational numbers are those numbers that we can express as the ratio of two integers, say a/b, where the denominator is not equal to zero. The Greek Pythagorean school of mathematicians discovered another fundamental type of number. To their dismay, they found that some quantities cannot be expressed as the ratio of two integers. The first such number they found was $\sqrt{2}$ when they tried to find the number representing the length of the diagonal through a unit square (a square with sides of length one). They showed that no such rational number exists. Soon other examples of these irrational numbers were found. Curiously, π, one of the most intensively studied of nature's fundamental constants, was not shown to be irrational until the eighteenth century. The combined rational and irrational numbers make up the set of real numbers.

David D. Clark, 246 S. Fraser St. #2, State College, PA 16801.

Complex Numbers

The real numbers are themselves a subset of an infinitely larger set of numbers called complex numbers. To understand what complex numbers are, consider the following equation:

$$x^2 + x + 1 = 0 \qquad (1)$$

Equation (1) looks innocuous enough. But, when we apply the quadratic equation in an attempt to solve the equation for the value of x, we obtain:

$$x = 1/2 \pm 1/2\sqrt{3}\sqrt{-1} \qquad (2)$$

What is $\sqrt{-1}$? Good question. Clearly, no real number will work since no real number squared equals -1. Mathematicians have created a special number, usually denoted i or j, to symbolize $\sqrt{-1}$. Thus Equation (2) represents:

"The notation a + bi is a rather unfortunate historical accident. It is better to think of a complex number as an ordered pair."

$$x = 1/2 + 1/2\sqrt{3}\,i \qquad (3)$$
$$x = 1/2 - 1/2\sqrt{3}\,i$$

Such numbers are called complex numbers and consist of a real part (1/2 in Equation (3)) and an imaginary part ($1/2\sqrt{3}\,i$ and $-1/2\sqrt{3}\,i$). They are commonly represented as:

$$x = a + bi \qquad (4)$$

where a and b are real numbers. a is called the real part and bi is called the imaginary part. The real numbers are all of those complex numbers where $b = 0$. Complex numbers where $a = 0$ are called pure imaginaries. Complex

numbers are equal if and only if both the real parts and the imaginary parts are equal. Complex numbers that differ only in the sign of their imaginary parts are called conjugates. An interesting and useful property of conjugates is that, when two conjugates are multiplied, the result is always real.

The notation of Equation (4) is a rather unfortunate historical accident. It tends to confuse people on their first exposure to complex numbers. We can never actually add the real and imaginary parts as the plus sign suggests. It is better to think of a complex number simply as an ordered pair (a, bi). This concept immediately suggests a geometric representation. A plane defined by a real axis and an orthogonal imaginary axis is called an Argand plane after the Swiss-French mathematician Jean Robert Argand (1768 – 1822). The geometric interpretation of complex numbers is also attributed independently to the Norwegian surveyor Caspar Wessel (1745 – 1818). Scholars have suggested that complex numbers be renamed "normal" numbers since, in the geometric representation of such quantities, the imaginary axis is normal to the real axis. Figure 1 (below) shows how the number $3 + 4i$ would be represented on such a plane. Because it is often convenient to think of complex numbers simply as points on a complex plane, we will use the terms *number* and *point* interchangeably in the following discussion.

We can also represent complex numbers with polar coordinates. In polar coordinates, a point is defined in reference to a fixed line and a point on that line, called the origin or pole. In this system, complex numbers are written as:

$$x = r(\cos\theta + i \sin\theta) = r \text{ Cis}\theta \qquad (5)$$

where r, called the radius or modulus, represents the distance of the point from the origin, and θ, called the amplitude, represents an angle of rotation from the reference line. Figure 2 (below) demonstrates how to graph the number 5 Cis 0.9273. The two forms may be interconverted easily. The polar coordinates for $x = a + bi$ are:

$$r = \sqrt{a^2 + b^2} \qquad (6)$$
$$\theta = \tan^{-1}(a/b)$$

and the rectangular coordinates for $r = \text{Cis}\theta$ are:

$$|a| = r \cos\theta \qquad (7)$$
$$|b| = r \sin\theta$$

Since we calculate only the absolute values of a and b when converting from polar form, we must determine the actual signs of the two quantities from the quadrant in which the point falls. The quadrants are numbered in increasing order starting with quadrant I as the upper righthand part of the graph and proceeding counterclockwise. The table (above) shows how the signs of a and b are related to the quadrant of the plane in which the point appears.

Quadrant	Sign of a	Sign of b
I	+	+
II	−	+
III	−	−
IV	+	−

The signs of the real (a) and imaginary (b) parts of complex numbers as a function of the quadrant in which the point appears.

Table

So, after a little fiddling on a calculator, we see that:

$$3 + 4i = 5 \text{ Cis } 0.9273 \qquad (8)$$

and that Figures 1 and 2 represent the same number in the two coordinate systems.

Complex Arithmetic

We can do arithmetic with complex numbers just as we do with the more familiar real numbers. The four arithmetic operations of addition, subtraction, multiplication, and division are very simple with complex numbers. As shown below, for multiplication and division we make use of the fact that $i^2 = -1$.

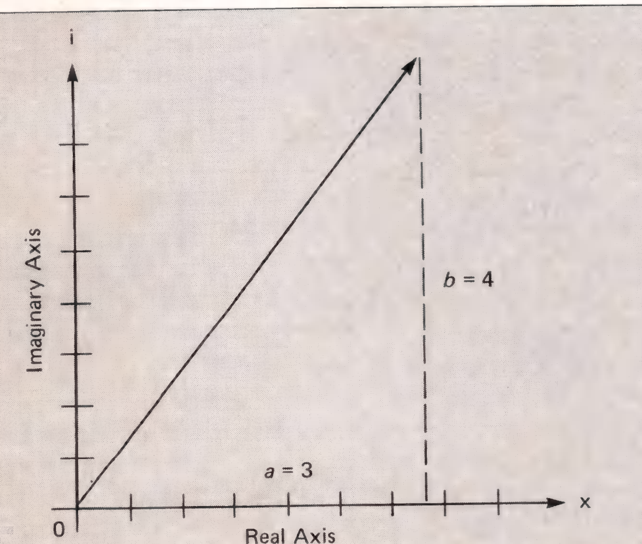


Figure 1.
Geometric representation of the complex number 3 + 4*i* in rectangular coordinates on the Argand plane.

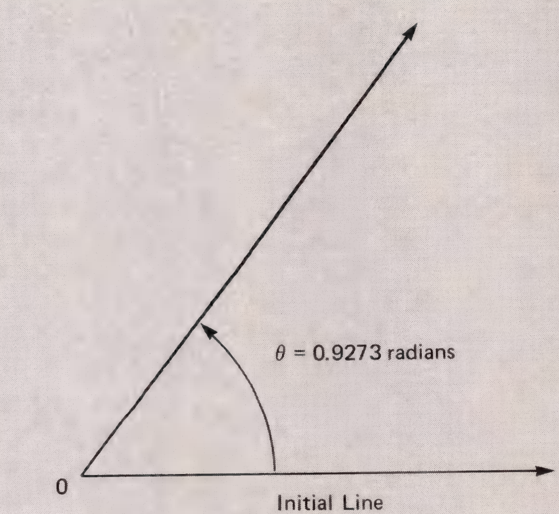


Figure 2.
Geometric representation of the complex number 5 Cisθ graphed in polar coordinates.

Addition

$$(a + bi) + (c + di) = (a + c) + (b + d)i \quad (9)$$

Subtraction

$$(a + bi) - (c + di) = (a - c) + (b - d)i \quad (10)$$

Multiplication

$$(a + bi) \times (c + di) = a(c + di) + b(c + di)i = (ac - bd) + (bc + ad)i \quad (11)$$

Division

$$\frac{a + bi}{c + di} = \frac{(a + bi) \times (c - di)}{(c + di) \times (c - di)} \quad (12)$$

$$= \frac{ac + bd}{c^2 + d^2} + \left[\frac{bc - ad}{c^2 + d^2}\right] i$$

Complex Functions

The calculation of complex functions is not much more difficult than doing complex arithmetic. One of the simplest complex functions is raising a complex number to an integer power. Polar notation makes the calculation easy:

$$(r \text{ Cis}\theta)^n = r^n \text{ Cis } n\theta \quad (13)$$

To raise e, the base of the natural logarithms, to a complex power:

$$e^{a + bi} = e^a e^{bi} \quad (14)$$

where, by Euler's formula:

$$e^{bi} = \cos b + i \sin b = \text{Cis } b \quad (15)$$

We can confirm the veracity of this formula by expanding the exponential in a Taylor series.

Taking the natural logarithm of a complex number is also fairly straightforward. If θ is restricted such that $-\pi < \theta \leqq \pi$ and $x = r \text{ Cis}\theta$:

$$\ln(x) = \ln(r) + i\theta \quad (16)$$

To raise a complex number to a complex power, we form an analogy to the operation with real numbers. For real numbers:

$$x^y = e^{y \ln(x)} \quad (17)$$

If x and y are complex:

$$x^y = e^{y \ln(r)} e^{iy\theta} \quad (18)$$

Equation (15) suggests a way to calculate the sine and cosine of complex numbers. If x is complex:

$$e^{xi} = \cos x + i \sin x \quad (19)$$

$$e^{-xi} = \cos x - i \sin x \quad (20)$$

Adding Equations (19) and (20) and rearranging, we arrive at:

$$\cos x = \frac{e^{xi} + e^{-xi}}{2} \quad (21)$$

Subtracting Equation (20) from Equation (19) and rearranging yields:

$$\sin x = \frac{e^{xi} - e^{-xi}}{2i} \quad (22)$$

We can now perform a reasonable number of operations with complex numbers. We can build up additional simple functions by proper sequences of the operations described.

Applications

So now that we can do all this wonderful math with complex numbers, what good are they? You might not think so, but complex numbers have a variety of uses. As implied near the beginning of this article, complex numbers have a fundamental importance in the field of mathematics because the roots of polynomial equations often contain imaginary terms. Also, almost all wave phenomena are described with the aid of complex numbers, from signal analysis in electronic circuits to the wave functions used by chemists and physicists to describe the nature of matter at the atomic and molecular level.

Applications familiar to most people interested in electronics include the design and description of filters and reactive and resonant circuits. Phasor diagrams represent the real, resistive parts of the circuit, while the imaginary axis shows the imaginary or reactive part. Such a diagram is very similar to the Argand plane.

When Ohm's law is generalized to AC circuits, it has the conventional form, familiar from DC circuit analysis:

$$I = V/Z$$

where I is the current, V is the voltage, and Z is the impedance. However, the impedance varies depending upon the type of device (resistor, capacitor, or inductor) and the frequency of the signal applied to the circuit:

resistor	$Z_R = R$
capacitor	$Z_C = -i/\omega C$
inductor	$Z_L = i\omega L$

where $\omega = 2\pi f$ and f is the frequency of the applied signal in cycles per second. R is the resistance in ohms, C is the capacitance in farads, and L is the inductance in henrys.

In my own work, I use complex numbers almost daily because of their presence in the Fourier transform. The Fourier transform is most commonly used to convert data in the form of intensity vs. time into frequency vs. time data. I use an instrument called an NMR (Nuclear Magnetic Resonance) spectrometer. This instrument detects the "flipping" of nuclear spin states. This type of experiment originally involved subjecting a sample to a constant radio frequency field and a varying strong magnetic field or a constant magnetic field and varying radio frequency. Varying one of these fields took a lot of time. Modern instruments use the constant magnetic field of a superconducting magnet and a brief, intense pulse of radio energy. Since a square wave, like the applied pulse, can be built up from a series of sine waves of different frequencies, the pulse experiment is equivalent to hitting the sample with all of those frequencies at once. The FID (Free Induction Decay) is recorded as a function of time after the pulse. This data is then transformed into the frequency domain. The peaks in the transformed signal represent the frequencies at which nuclear spins flipped.

Fourier transforms are used in all types of signal analysis applications, such as digital filtering and spectral analysis. The DFT (Discrete Fourier Transform) of $x(n)$ is defined as:

$$X(k) = \sum_{n=0}^{N-1} x(n) \exp(i2\pi nk/N)$$
$$k = 0,1,\ldots,N-1$$

THE PROGRAMMER'S SHOP™

helps compare, evaluate, find products. Straight answers for serious programmers.

SERVICES

- **Programmer's Referral List**
- **Compare Products**
- **Help find a Publisher**
- **Evaluation Literature free**
- **BULLETIN BOARD - 7 PM to 7 AM 617-826-4086**
- **Dealer's Inquire**
- **Newsletter**
- **Rush Order**
- **Over 300 products**

PUBLIC DOMAIN Research - Free

6 months paid research gives you leverage, learning. We found, combined, added to the best. All run, have source in C or ASM. Order $150 + get one free: Database, Editors, Modems, MSDOS RAMdisks & utils, Games in C.

RECENT DISCOVERIES

BRIEF Programmer's Editor is terrific for PCDOS. Worth effort to switch. Powerful, flows well. Macros, reconfigure. Contest $195

"C" LANGUAGE

	LIST PRICE	OUR PRICE
MSDOS: C86-8087, reliable	$395	call
Desmet with debugger	159	145
Lattice 2.1 - improved	500	call
Microsoft C 2.x	500	349
Williams - NEW, debugger	500	call
CPM80' Ecosoft C-now solid, full	250	225
BDS C - solid value	150	125
MACINTOSH: In stock!	NA	385
LINT-like for C86, Lattice	NA	400
Source-level Profiler: C86, Lat	NA	150
MACINTOSH: Full, ASM	NA	385

Compare, evaluate, consider other Cs

BASIC

	ENVIRONMENT		
BASCOM-86 - MicroSoft	8086	395	279
BASIC Dev't System	PCDOS	79	72
BASICA Compiler - BetterBASIC - 640K	PCDOS	—	199
CB-86 - DRI	CPM86	600	439
Prof. BASIC Compiler	PCDOS	345	325
MACINTOSH COMPILER with BASICA syntax	MAC	NA	325

Ask about ISAM, other addons for BASIC

FEATURES

Graphic C with full source - scientific plots, 4096 res. Optional 8087.C86, Desmet $195

PROLOG86 Interpreter for MSDOS includes tutorials, reference and good examples. Learn in first few hours. For Prototyping, Natural Language or AI. $125.

EDITORS Programming

BRIEF - Intuitive, flexible	PCDOS	NA	195
C Screen with source	8080/86	NA	75
FINAL WORD - for manuals	8080/86	300	215
MINCE - like EMACS	CPM, PCDOS	175	149
PMATE - powerful	CPM	195	175
	8086	225	195
VEDIT - full, liked	CPM, PCDOS	150	119
	8086	200	159

UNIX PC

COHERENT - for "C" users	PClike	$500	475
VENIX - "true V7" w/FTN	PClike	800	775
XENIX - "true S3" - rich	PC	1350	1285

Ask about run-times, applications, DOS compatibility, other alternatives UNIX is a trademark of Bell Labs

LANGUAGE LIBRARIES

C to dBASE interface	8080/85	$150	$140
C Tools 1 - String, Screen	PCDOS	NA	115
C Tools 2 - OS Interface	PCDOS	NA	92
GRAPHICS: GSX - 80	CPM80	NA	75
HALO - fast, full	PCDOS	200	175
Greenleaf for C - full	PCDOS	NA	165
ISAM: C Index + -no royalties	MSDOS	NA	400
BTRIEVE - many languages	PCDOS	245	215
PHACT - with C	PCDOS	NA	250
PASCAL TOOLS - Blaise	PCDOS	NA	115
SCREEN: PANEL-86 - many languages	PCDOS	295	265
WINDOWS for C	PCDOS	NA	139

Ask about many others for FTN, BASIC, PASCAL, C-ISAM, Screen, Stat, Graphics.

PASCAL

	ENVIRONMENT	LIST PRICE	OUR PRICE
PASCAL MT + 86	CPM86/IBM	$400	$279
MS PASCAL 86	MSDOS	300	215
PASCAL 64 - nearly full	COM 64	99	89

OTHER PRODUCTS

AKA ALIAS - improve DOS	PCDOS	NA	60
Assembler & Tools - DRI	8086	200	159
CODESMITH-86 - debug	PCDOS	149	139
Disk Mechanic - rebuild	MSDOS	70	65
IQ LISP - full 1000K RAM	PCDOS	175	call
MBP Cobol-86 - fast	8086	750	695
MicroPROLOG	PCDOS	NA	265
Microsoft MASM-86	MSDOS	100	85
MS Fortran - improvements	MSDOS	350	255
Multilink-Multitasking	PCDOS	295	265
PL/1-86	8086	750	495
PLINK-86 - overlays	8086	350	315
Polylibrarian - thorough	MSDOS	99	89
PROFILER-86 - easier	MSDOS	NA	125
PROFILER - flexible	MSDOS	NA	175
Programmers Tlkt w/source	8086	NA	135
READ CPM86 from PCDOS	PCDOS	NA	55
READ PCDOS on an IBM PC	CPM86	NA	55
TRACE86 debugger ASM	MSDOS	125	115

Note: All prices subject to change without notice. Mention this ad. Some prices are specials.

Ask about COD and POs.
All formats available.

Call for a catalog, literature, and answers

800-421-8006

THE PROGRAMMER'S SHOP™

128-D Rockland Street, Hanover, MA 02339

Visa Mass: 800-442-8070 or 617-826-7531 MasterCard

Circle **no. 53** on reader service card.

PROLOG-86™

Learn Fast, Experiment

1 or 2 pages of PROLOG would require 10 or 15 pages in "C."

Be familiar in **one evening.** In a **few days** enhance artificial intelligence programs included like:

- **an Expert System**
- **Natural Language** **(generates dBASE display)**

Intro Price: $125 for PCDOS, CPM-86.
Full Refund if not satisfied.

CONTEST: "Artificial Intelligence Concepts"

$1,000 Prize, Recognition for applications in PROLOG-86™ that teach, are clear, illustrate. Call for details. Deadline 11/31/84

SOLUTION SYSTEMS™
45-D Accord Park, Norwell, MA 02061
617-871-5435

Circle **no. 84** on reader service card.

C Helper™

FIRST-AID FOR C PROGRAMS

Save time and frustration when analyzing and manipulating C programs. Use C HELPER's UNIX-like utilities which include:

DIFF and **CMP** – for "intelligent" file comparisons.
XREF – cross references variables by function and line.
C Flow Chart – shows what functions call each other.
C Beautifier – make source more regular and readable
GREP – search for sophisticated patterns in text.

There are several other utilities that help with converting from one C compiler to another and with printing programs.

C Helper is written in portable C and includes both full source code and executable files for $135 for MS-DOS, CPM-80 or CPM-86. Use VISA, Master Card or COD.

Call: **617-659-1571**

Solution Systems™
335-D Washington Street
Norwell, MA 02061

Circle **no. 85** on reader service card.

The inverse DFT is:

$$x(n) = 1/N \sum_{k=0}^{N-1} X(k) \exp(i2\pi nk/N);$$
$$n = 0,1,\ldots,N-1$$

We can also use the Fourier transform to analyze an interesting signal based on a complex function, the frequency-modulated wave related to FM radio transmission:

$$x(k) = \exp[i(ak + b \cos ck)]$$

Analysis of this type of signal reveals a sharp central peak and a number of sidebands whose position and intensities are influenced by the values of the constants a, b, and c.

Complex numbers are also used extensively in the study of electromagnetic phenomena. Calculation of radiated power, radiation pressure, and design of waveguides are good examples of this use.

Computer Implementation

Given all that you can do with complex numbers, the next step is getting your computer to help. It's easy to do in Fortran since COMPLEX is one of the fundamental data types available. In addition, the Fortran library has a large number of functions that operate with COMPLEX variables. However, I have an intense dislike of Fortran, and the Fortran compilers I have available for my microcomputer do not support the COMPLEX data type. Since I *do* like Pascal, I wrote the routines I wanted in that language.

Listing One (page 36) is a UCSD Pascal unit called Complex Arithmetic that implements the operations discussed above. First, a public data type called Complex is declared. Complex numbers will be represented by this data type in programs that make use of the unit. It consists of a record containing two fields. The Re field represents the real coefficient, while the Im field represents the imaginary coefficient. All complex arguments and results are in rectangular coordinates, although some of the calculations use polar coordinates.

This approach has some disadvantages. The primary disadvantage is that you must build up equations from successive procedure and function calls. This is because Pascal does not allow functions to return composite values; it is not possible to write routines that return results of data type Complex. It would also be nice if you could overload the arithmetic operators to use Complex data types. You can do it in Ada, but that's another story.

The four arithmetic operations are straightforward implementations of Equations (9) through (12). The Polar routine, which converts rectangular coordinates to polar coordinates, requires some explanation. Because of the way the p-System handles certain exceptional numerical conditions, a few precautions had to be taken when coding the routine. The first two conditionals are present to prevent underflow when squaring the real and imaginary parts of the argument. Squaring numbers very close to zero can produce a number too close to zero for the computer to represent. For example, squaring 10^{-20} yields 10^{-40}, which is too small for many systems to handle. This is called underflow. When it happens on my machine, a runtime error occurs and the program is aborted. The two conditional statements detect such a situation before the damage is done and prevent an error by setting the coefficient to zero (since Arg is passed by value, the reassignment only has effect while within procedure Polar). Squaring zero does not cause any problems.

The calculation of the amplitude is a little trickier. While the ATan function is very robust (you can't hurt it, no matter what argument you hit it with), division is not quite so resilient. Division can cause problems in a couple of ways. First, division by zero is certain death. Second, dividing a very large number by a very small number can cause overflow. For example, $10^{20}/10^{-20} = 10^{40}$. There is a way to detect such a situation before it happens. Since:

$$\mathrm{abs}(b)/\mathrm{abs}(a) = e^{(\ln(\mathrm{abs}(b)) - \ln(\mathrm{abs}(a)))}$$

we can check the relative magnitudes of a and b before doing the actual division. If an infinite quantity (to the computer) is about to be calculated, signs are checked and an appropriate value is assigned directly. (The ATan function approaches $\pi/2$ as its argument increases without bound in the positive direction. It tends to $-\pi/2$ as the argument decreases toward negative infinity.) All this checking makes Polar an expensive procedure in terms of execution time. If you don't need all of these safeguards, you can reduce the procedure to the two statements:

```
Modulus :=
    Sqrt(Sqr(Arg.Re) + Sqr
      (Arg.Im));

Amplitude :=
    ATan(Arg.Im/Arg.Re);
```

(A note to p-System users: Some early versions of the interpreter have Ln functions that do not always return a result of the correct sign.)

The CToPower, CExp, and CLn procedures are almost straight transliterations of Equations (13) through (16). The procedures CToC, CSin, and CCos are examples of how equations involving complex numbers can be built up one operation at a time. Other functions, such as taking complex powers of real numbers, can be built up by calling these procedures with appropriate arguments. For example, to raise 5.3 to the $7.0 + 3.0i$ power, use CToC with Arg1 $= 5.3 + 0.0i$ and Arg2 $= 7.0 + 3.0i$.

Listing Two (page 42) is a typical application for complex numbers, a test program for FFT (Fast Fourier Transform) functions. The program generates a function with an analytically known DFT, transforms it, compares the calculated transform with the analytic transform, then reverses the process. The calculated inverse transform is then compared with the original function. The function generated is:

$$x(n) = Q^n \quad n = 0,1,\ldots,N-1$$

and its DFT is:

$$X(k) = (1 - Q^N)/(1 - QW^k)$$
$$k = 0,1,\ldots,N-1$$

where $W = \exp(-i2\pi/N)$ and Q is the complex constant $0.9 + 0.3i$. In the program, $N = 64$ so the transform is performed on 64 complex points.

The procedure that performs the actual calculation of the FFT is "included" from the file CURFFT.TEXT,

shown in Listing Three (page 46). That way you can test several different FFT routines using the same main program. Just create your new Fourea procedure and stuff it in a file called CURFFT.TEXT. As stated in the listing, this is not a particularly efficient FFT procedure, but it has the virtue of simplicity (for an FFT calculation anyway) and illustrates a "real life" application of complex numbers.

The output at the end of Listing Three shows the results of running the FFT testing program. A large difference between the calculated transforms and the analytically known values for the transforms indicates that there is probably an error in the Fourea procedure being tested.

Summary

Complex numbers have been introduced in relation to the more familiar rational and real number systems. Complex numbers consist of a real part and an imaginary part. The imaginary part is so named because one of its factors is $\sqrt{-1}$, more commonly denoted simply as i. A geometrical interpretation of such numbers has been described in which complex numbers represent points on a complex plane, called an Argand plane. The points can be plotted in rectangular or polar coordinates, and the two coordinate systems can be easily interconverted.

Complex numbers have an associated algebra, just like the rational and real numbers. The operations of addition, subtraction, multiplication, and division, as well as the calculation of some common functions of complex numbers, have been explained. A UCSD Pascal unit called Complex Arithmetic, which implements these calculations in a specific programming language, has been described, as well as an example program to calculate fast Fourier transforms using complex arithmetic.

In addition to signal analysis and Fourier analysis, complex numbers are used in the mathematics of electronics (e.g., filters and reactive and resonant circuits). Phasor diagrams consist of a real part, the resistive part, and an imaginary part, the reactive part. The wave functions used by chemists and physicists in the study of quantum mechanics also make use of the properties of complex numbers. And now, so can you.

References

1. Crandall, R. E., *Pascal Applications for the Sciences* (John Wiley & Sons, Inc., 1984). A very good "how to" book on the use of computers in the sciences. It contains a large number of small Pascal programs for solving problems in several areas of the physical and biological sciences.
2. Dence, J. B., *Mathematical Techniques in Chemistry* (John Wiley & Sons, Inc., 1975). This book is invaluable to chemists. It describes a wide range of mathematical techniques as applied to chemical problems.
3. Digital Signal Processing Committee of the IEEE Acoustics, Speech, and Signal Processing Society, *Programs for Digital Signal Processing* (IEEE Press, 1979). This book contains lots of very fancy programs for use in various areas of signal processing. The discussion of the theory of such calculations is minimal. All the programs are in Fortran. The program in Listings Two and Three was derived from a simple demonstration program in this book.
4. Horowitz, P., and W. Hill, *The Art of Electronics* (Cambridge University Press, 1980). This book contains a good introduction to the use of complex quantities in the design and description of electronic signal processing circuitry such as filters and resonant and power circuits. It is also an excellent introduction to electronics in general.
5. Kousourou, G., S. N. Sonsky, and F. Scalzo, *An Introduction to Technical Mathematics with Computing* (Petrocelli Books, Inc., 1979). A textbook of elementary mathematics required in most technical work. It has a good introduction to complex numbers and their manipulation. Lots of problems and worked examples.
6. Lorrain, P., and D. R. Corson, *Electromagnetic Fields and Waves* (W. H. Freeman and Company, San Francisco, 1970). Some very heavy reading on the theoretical aspects of electromagnetic phenomena. Complex quantities permeate the mathematics of this book.

DDJ

(Listings begin on page 36)

Calculations (Text begins on page 30)

(Text begins on page 30)

Listing One

```
{xL PRINTER:}
UNIT ComplexArithmetic;
{ data type and procedures for performing complex arithmetic }

  INTERFACE

    TYPE  Complex = RECORD
                      Re : Real;
                      Im : Real
                    END { of RECORD };

    PROCEDURE CAdd(VAR Result: Complex; Argl, Arg2: Complex);
    { adds "Argl" and "Arg2" and returns the sum in "Result" }

    PROCEDURE CSub(VAR Result: Complex; Argl, Arg2: Complex);
    { subtracts "Arg2" from "Argl" and returns the difference in "Result" }

    PROCEDURE CMult(VAR Result: Complex; Argl, Arg2: Complex);
    { multiplies "Argl" and "Arg2" and returns the product in "Result" }

    PROCEDURE CDiv(VAR Result: Complex; Argl, Arg2: Complex);
    { divide "Argl" by "Arg2" and returns the quotient in "Result" }

    PROCEDURE Polar(Arg: Complex; VAR Modulus, Amplitude: Real);
    { converts a Complex number "Arg" in rectangular form to Polar form }

    PROCEDURE CToPower(VAR Result: Complex; Arg: Complex; Power: Integer);
    { raises "Arg" to the positive integral "Power" and returns the
      answer in "Result" }

    PROCEDURE CExp(VAR Result: Complex; Arg: Complex);
    { raises e to the "Arg" and returns the answer in "Result" }

    PROCEDURE CLn(VAR Result: Complex; Arg: Complex);
    { takes the natural logarithm of "Arg" and returns the answer in
      "Result" }

    PROCEDURE CToC(VAR Result: Complex; Argl, Arg2: Complex);
    { raise Complex number "Argl" to Complex Power "Arg2" }

    PROCEDURE CSin(VAR Result: Complex; Arg: Complex);
    { takes the sine of "Arg" and returns it in "Result }

    PROCEDURE CCos(VAR Result: Complex; Arg: Complex);
    { takes the cosine of "Arg" and returns it in "Result }

  IMPLEMENTATION

    CONST LN_MAX_REAL = 87.49823353; { ln(1.0e38) }
          PI_OVER_2   = 1.570796327; { pi/2.0 }
          CLOSEST     = 1E-19;       { sqrt(1.0e-38 }
  PROCEDURE CAdd{VAR Result: Complex; Argl, Arg2: Complex};

  BEGIN { CAdd }
    Result.Re := Argl.Re + Arg2.Re;
```

(Continued on page 38)

Listing One

```
   Result.Im := Arg1.Im + Arg2.Im
 END { of CAdd };


 PROCEDURE CSub(VAR Result: Complex; Arg1, Arg2: Complex);

 BEGIN { CSub }
   Result.Re := Arg1.Re - Arg2.Re;
   Result.Im := Arg1.Im - Arg2.Im
 END { of CSub };


 PROCEDURE CMult(VAR Result: Complex; Arg1, Arg2: Complex);

 BEGIN { CMult }
   Result.Re := Arg1.Re*Arg2.Re - Arg1.Im*Arg2.Im;
   Result.Im := Arg1.Im*Arg2.Re + Arg1.Re*Arg2.Im
 END { of CMult };


 PROCEDURE CDiv(VAR Result: Complex; Arg1, Arg2: Complex);

 VAR   Denom: Real;

 BEGIN { CDiv }
   Denom := Sqr(Arg2.Re) + Sqr(Arg2.Im);
   Result.Re := (Arg1.Re*Arg2.Re + Arg1.Im*Arg2.Im)/Denom;
   Result.Im := (Arg1.Im*Arg2.Re - Arg1.Re*Arg2.Im)/Denom
 END { of CDiv };


 PROCEDURE Polar(Arg: Complex; VAR Modulus, Amplitude: Real);

 BEGIN { Polar }
   WITH Arg DO BEGIN
        IF Abs(Re) < CLOSEST THEN
           Re := 0.0;
        IF Abs(Im) < CLOSEST THEN
           Im := 0.0;
        Modulus := Sqrt(Sqr(Re) + Sqr(Im));
        IF Im = 0.0 THEN
           Amplitude := 0.0
        ELSE IF Re = 0.0 THEN
           IF Im > 0.0 THEN
              Amplitude := PI_OVER_2
           ELSE
              Amplitude := -PI_OVER_2
        ELSE IF (Ln(Abs(Im)) - Ln(Abs(Re)) > LN_MAX_REAL) THEN
           IF Re > 0.0 THEN

           IF Im > 0.0 THEN
              Amplitude := PI_OVER_2
           ELSE
              Amplitude := -PI_OVER_2
        ELSE
           IF Im > 0.0 THEN
              Amplitude := -PI_OVER_2
           ELSE
              Amplitude := PI_OVER_2
    ELSE
```

(Continued on page 40)

WIZARD C

Fast compiles, fast code and great diagnostics make Wizard C unbeatable on MSDOS. Discover the powers of Wizard C:

- ALL UNIX SYSTEM III LANGUAGE FEATURES.
- UP TO A MEGABYTE OF CODE OR DATA.
- SUPPORT FOR 8087 AND 80186.
- FULL LIBRARY SOURCE CODE, OVER 200 FUNCTIONS.
- CROSS-FILE CHECKS OF PARAMETER PASSING.
- USES MSDOS LINK OR PLINK-86.
- CAN CALL OR BE CALLED BY PASCAL ROUTINES.
- IN-LINE ASSEMBLY LANGUAGE.
- 240 PAGE MANUAL WITH INDEX.
- NO LICENSE FEE FOR COMPILED PROGRAMS.

The new standard for C Compilers on MSDOS!

Only $450

WSS

For more information call (617) 641-2379
Wizard Systems Software, Inc.
11 Willow Ct., Arlington, MA 02174
Visa/Mastercard accepted

Circle **no. 77** on reader service card.

Now Your Computer Can See!

$295.00*

A total imaging system complete and ready for plug-and-go operation with your personal computer.

MicronEye™ "Bullet"

The MicronEye™ offers selectable resolution modes of 256 x 128 and 128 x 64 with operating speeds up to 15 FPS. An electronic shutter is easily controlled by software or manual functions, and the included sample programs allow you to continuously scan, freeze frame, frame store, frame compare, print and produce pictures in shades of grey from the moment you begin operation.

Only the MicronEye™ uses the revolutionary IS32 OpticRAM™ image sensor for automatic solid state image digitizing, with capability for greytone imaging through multiple scans. And with these features, the MicronEye™ is perfectly suited for graphics input, robotics, text and pattern recognition, security, digitizing, automated process control and many other applications.

The MicronEye™ is available with immediate delivery for these computers: Apple II, IBM PC, Commodore 64 and the TRS-80CC (trademarks of Apple Computer Inc., International Business Machines, Commodore Corp., and Tandy Corp. respectively).

Phone for MicronEye™ information on the Macintosh, TI PC and RS232 (trademarks of Apple Computer Inc. and Texas Instruments respectively.)

*(Add $10.00 for shipping and handling [Federal Express Standard Air]; residents of the following states *must add sales tax*: AK, AZ, CA, CO, CT, FL, GA, IA, ID, IL, IN, LA, MA, MD, ME, MI, MN, NC, NE, NJ, NY, OH, PA, SC, TN, TX, UT, VA, VT, WA, WI.)

MICRON TECHNOLOGY, INC.

VISION SYSTEMS
2805 East Columbia Road
Boise, Idaho 83706
(208) 383-4106
TWX 910-970-5973

Circle **no. 44** on reader service card.

Use ALL the Power of Your MS-DOS, IBM PC-DOS, or CP/M-80 System with UNIX-Style Carousel Tools


```
ch "CP/M" "MS-DOS" <doc >newdoc
diff newdoc doc | more
ed newdoc
kwic newdoc | sortmrg | uniq | unrot >index
make -f makdoc ndx
```

Carousel Tools and Carousel ToolKits are trademarks of Carousel MicroTools, Inc. CP/M is a trademark of Digital Research; IBM is a trademark of International Business Machines; MS is a trademark of MicroSoft; UNIX is a trademark of Bell Laboratories.

CAROUSEL TOOLS are a proven set of over 50 programs designed to be used with pipes, redirected I/O and scripts. In the style of UNIX each Tool does one thing well, and the Tools can be used together to do more complex tasks.

YOU ACCOMPLISH MORE using Carousel Tools: better programming and documentation support, simpler data and file housekeeping, more general file handling.

TOOLS FOR PC/MS-DOS 2.x AND CP/M-80 are available now. The DOS ToolKit is $149. The CP/M ToolKit is $249 and includes a *shell* to provide pipes, redirected I/O, and scripts. Source code is available for $100 more.

ORDER YOUR TOOLKIT TODAY.

CALL OR WRITE:

CAROUSEL MICROTOOLS, INC.

609 Kearney Street, El Cerrito, CA 94530 (415) 528-1300

Circle **no. 10** on reader service card.

Listing One

```
           Amplitude := ATan(Im/Re)
   END { of WITH Arg }
 END { of Polar };


 PROCEDURE CToPower{VAR Result: Complex; Arg: Complex; Power: Integer}

 VAR    I: Integer;
        Modulus, Amplitude, NewMod, PowAmp: Real;

 BEGIN { CToPower }
   IF Power = 0 THEN BEGIN
      Result.Re := 1.0;
      Result.Im := 0.0
   END
   ELSE BEGIN
        Polar(Arg, Modulus, Amplitude);
        NewMod := 1;
        IF Power > 0 THEN
           FOR I := 1 TO Power DO NewMod := NewMod*Modulus
        ELSE
           FOR I := 1 TO Abs(Power) DO NewMod := NewMod/Modulus;
        PowAmp := Power*Amplitude;
        Result.Re := NewMod*Cos(PowAmp);
        Result.Im := NewMod*Sin(PowAmp)
   END
 END { of CToPower };


 PROCEDURE CExp{VAR Result: Complex; Arg: Complex};

 VAR    Expo: Real;

 BEGIN { CExp }
   Expo := Exp(Arg.Re);
   Result.Re := Expo*Cos(Arg.Im);
   Result.Im := Expo*Sin(Arg.Im)
 END { of CExp };


 PROCEDURE CLn{VAR Result: Complex; Arg: Complex};

 VAR    Modulus, Amplitude: Real;

 BEGIN { CLn }
   Polar(Arg, Modulus, Amplitude);
   Result.Re := Ln(Modulus);
   Result.Im := Amplitude
 END { of CLn };


 PROCEDURE CToC{VAR Result: Complex; Arg1, Arg2: Complex};

 VAR    LogPart, Expo: Complex;

 BEGIN { CToC }
   CLn(LogPart, Arg1);
   CMult(Expo, Arg2, LogPart);
   CExp(Result, Expo)
 END { of CToC };
```

```
PROCEDURE CSin(VAR Result: Complex; Arg: Complex);

VAR Expl, Exp2, Partl, Part2, Sum, Divisor: Complex;

BEGIN { CSin }
  Expl.Re := -Arg.Im;          { z*i }
  Expl.Im := Arg.Re;
  CExp(Partl, Expl);           { exp(zi) }
  Exp2.Re := Arg.Im;           { -z*i }
  Exp2.Im := -Arg.Re;
  CExp(Part2, Exp2);           { exp(-zi) }
  CSub(Sum, Partl, Part2);     { exp(zi) - exp(-zi) }
  Divisor.Re := 0.0;
  Divisor.Im := 2.0;
  CDiv(Result, Sum, Divisor) { (exp(zi) - exp(-zi))/(2i) }
END { of CSin };


PROCEDURE CCos(VAR Result: Complex; Arg: Complex);

VAR Expl, Exp2, Partl, Part2, Sum, Divisor: Complex;

BEGIN { CCos }
  Expl.Re := -Arg.Im;          { z*i }
  Expl.Im := Arg.Re;
  CExp(Partl, Expl);           { exp(zi) }
  Exp2.Re := Arg.Im;           { -z*i }
  Exp2.Im := -Arg.Re;
  CExp(Part2, Exp2);           { exp(-zi) }
  CAdd(Sum, Partl, Part2);     { exp(zi) + exp(-zi) }
  Divisor.Re := 2.0;
  Divisor.Im := 0.0;
  CDiv(Result, Sum, Divisor) { (exp(zi) + exp(-zi))/(2) }
END { of CCos };


BEGIN { ComplexArithmetic }
END { of ComplexArithmetic }.
```

End Listing One

(Listing two begins on next page)

Listing Two

```
Program FFTTest(Input, Output, Prn);

{*
**       This  program  generates  a  function  to test various fast Fourier
**  transform programs.  As currently implemented, the program compiles  the
**  routine  to  be  tested  in-line  with  the  test  program.    This   is
**  accomplished through the use of  the  "include"  file  mechanism  of the
**  compiler.  The  routine  to  be  tested  should  be  in  a  file  called
**  "CURFFT.TEXT".  The routine's heading should be as follows:
**
**       Procedure Fourea(    numPoints : Integer;
**                            whichWay  : Direction;
**                        Var f         : CmplxArray);
**
**  where the types Direction and CmplxArray are as declared below.
**
**       The program computes the function A^I, I = 0, 1, ... MAXPOINTS - 1.
**  The  discrete  Fourier transform of this function is also computed.  The
**  procedure Fourea is called to compute the fast  Fourier transform in the
**  forward direction.  This call is then  repeated,  but  for  the  reverse
**  direction.  The  theoretical  transform is then compared to the computed
**  transform and the  calculated  reverse  transform  is  compared  to  the
**  original function.  The maximum difference  of  each  comparison is then
**  printed.  A large maximum difference probably indicates a program error.
**
**       The  Unit  ComplexArithmetic  is  used to perform the arithmetic on
**  complex numbers required by the  algorithm.  The value of  MAXPOINTS  is
**  set  at  64 because it is also a power of 8.  This is  important because
**  the program is also intended to test a version of Fourea  optimized  for
**  data arrays of 512 complex points.
**
**  Based on a Fortran program by C. M. Rader
**  MIT Lincoln Laboratory, Lexington MA   02173
**
**  Written by David D. Clark
**  26-Mar-83
*}

Uses {$U MATH.LIBRARY} ComplexArithmetic;

Const MAXPOINTS   = 64;                 { size of the transformed array }
      TWOPI       = 6.2831853;          { 2 * pi }
      PRINTFILE   = 'PRINTER:';         { output device file name for results

Type  Direction   = (Forwurd, Reverse); { note the spelling }
      CharFile    = File of Char;
      CmplxArray  = Array [1..MAXPOINTS] of Complex;

Var   I           : Integer;            { general purpose index variable }
      DD,
      D1,
      D2,
      GG,
      G1,
      G2,
TwoPiDivMax : Real;                { will be TWOPI/MAXPOINTS }
OneZero,                           { 1.0 + 0.0*i (complex number 1) }
A,
D,
```

(Continued on page 44)

CP/M® Software

A>DBPACK: Information Manager -- Great for Mailing Lists, Form letters, Tabulation and organizing data. Supports query, sort/search on multiple keys, report generation and many other data base functions. $115/$25.

A>COMCOM: Communication program. Uploads/Downloads files, and more. $95/$15.

A>CPMCPM: Transfers files (any type) between CP/M computers with incompatible disks. $65/$10 includes copy for each computer.

A>FILER: Archives, Sorts and Catalogs files with substantial disk space savings. $49.

A>BASXREF: Alphabetizes and Cross-references variables vs. line numbers in BASIC programs. Simplifies program maintenance. $39.

A>UNERA: Recovers erased files. $29.

CP/M is a registered trademark of Digital Research, Inc.

* Available in most disk formats.
* Clearly written and indexed manuals included. Where two prices are quoted, second refers to manual only (creditable towards software).
* All packages returnable in 15 days.

COMPU-DRAW
1227 Goler House
Rochester, NY 14620
(716)-454-3188

MasterCard, Visa & Amex cards, PO's from recognized institutions and COD are welcome.

Circle **no. 13** on reader service card.

GGM — FORTH™ has HELP*
for Z80[1] using CP/M[2]

GGM—FORTH, a complete software system for real-time measurement and control, runs on any Z80 computer under CP/M using an extended fig-FORTH vocabulary.

GGM—FORTH features:

- Open multiple CP/M files, in any combination of direct-access and sequential-access, fully compatible with all CP/M utilities
- Char. in/out uses CP/M console, lister, file, or port
- On-line HELP* provides instant access to definitions in the run-time GGM—FORTH dictionary
- HELP* file is easily extended to include user definitions using HELP* utility
- HELP* is available during full-screen editing

Complete system and manuals	$150.
Manuals only:	$ 20.
Introductory System:	$ 35.

GGM SYSTEMS, INC. (617) 662-0550
135 Summer Ave., Reading, MA 01867

[1]Z80 is a trademark of Zilog, Inc.
[2]CP/M is a trademark of Digital Research, Inc.

Circle **no. 28** on reader service card.

UNPARALLELED
PERFORMANCE
and **PORTABILITY**
in an **ISAM** PACKAGE
at an **UNBEATABLE**
PRICE

c-tree™
BY FAIRCOM

2606 Johnson Drive
Columbia MO 65203
(314) 445-6833

The company that introduced micros to B-Trees in 1979 and created ACCESS MANAGER™ for Digital Research, now redefines the market for high performance, B-Tree based file handlers. With c-tree™ you get:

- complete C source code written to K & R standards of portability
- high level, multi-key ISAM routines and low level B-Tree functions
- routines that work with single-user and network systems
- no royalties on application programs

$395 COMPLETE

Specify format:
8″ CP/M® 5¼″ PC-DOS 8″ RT-11

for VISA, MC or COD orders, call toll free
1-800-232-3344

Access Manager and CP/M are trademarks of Digital Research, Inc.
c-tree is a trademark of FairCom.

© 1984 FairCom

Circle **no. 25** on reader service card.

Listing Two

```
        E,
        W,
        Tem,
        TemTem        : Complex;
        B,                                    { complex data arrays }
        C,
        Qb            : CmplxArray;
        PrnFile       : CharFile;             { print out file }


    Procedure PrintOut(    numPoints : Integer;
                       Var f         : CmplxArray;
                       Var dev       : CharFile    );
    {*
    **  Print out the numPoint members of complex array f on dev
    *}

    Var   i,
          j : Integer;

    Begin { PrintOut }
       WriteLn(dev);
       For i := 1 to numPoints Div 2 Do Begin
           j := 2*i - 1;
           WriteLn(dev, '(', j      : 2, ')', f[j    ].re: 14, f[j    ].im: 14,
                     '     (', (j + 1): 2, ')', f[j + 1].re: 14, f[j + 1].im: 14)
       End;
       WriteLn(dev);
       WriteLn(dev)
    End { of PrintOut };


{ now include the FFT procedure }
{$I CURFFT.TEXT}


Begin { FFTTest }

    { some initialization }
    OneZero.re := 1.0;  OneZero.im := 0.0;
    TwoPiDivMax := TWOPI/MAXPOINTS;
    W.re := Cos(TwoPiDivMax);  W.im := -Sin(TwoPiDivMax);
    A.re := 0.9;  A.im := 0.3;
    ReWrite(PrnFile, PRINTFILE);

    { Calculate and print function A^I, I := 0, 1, ... (MAXPOINTS - 1) }
    B[1] := OneZero;
    Qb[1] := OneZero;
    For I := 2 to MAXPOINTS Do Begin
        CToPower(B[I], A, I - 1);
        Qb[I] := B[I]

End;
WriteLn(PrnFile, 'Complex input sequence:');
PrintOut(MAXPOINTS, Qb, PrnFile);

{ Calculate and print the theorectical discrete Fourier transform }
CToPower(Tem, A, MAXPOINTS);
CSub(D, OneZero, Tem);
For I := 1 to MAXPOINTS Do Begin
```

```
    CToPower(TemTem, W, I - 1);
    CMult(Tem, A, TemTem);
    CSub(E, OneZero, Tem);
    CDiv(C[I], D, E)
End;
WriteLn(PrnFile, 'Theoretical discrete Fourier transform:');
PrintOut(MAXPOINTS, C, PrnFile);

{ Calculate and print the fast Fourier transform of the function }
Fourea(MAXPOINTS, Forwurd, B);
WriteLn(PrnFile, '"Fourea" generated discrete Fourier transform:');
PrintOut(MAXPOINTS, B, PrnFile);

{ Find the maximum difference }
DD := 0.0;
For I := 1 to MAXPOINTS Do Begin
    D1 := Abs(C[I].re - B[i].re);
    D2 := Abs(C[I].im - B[I].im);

    If D1 > DD Then
       If D2 > D1 Then
          DD := D2
       Else
          DD := D1
    Else If D2 > DD then
       DD := D2

End;

{ Calculate and print the inverse transform }
Fourea(MAXPOINTS, Reverse, C);
WriteLn(PrnFile, '"Fourea" generated inverse descrete Fourier transform:');
PrintOut(MAXPOINTS, C, PrnFile);

{ Find the maximum difference }
GG := 0.0;
For I := 1 to MAXPOINTS Do Begin
    G1 := Abs(Qb[I].re - C[i].re);
    G2 := Abs(Qb[I].im - C[I].im);

    If G1 > GG Then
       If G2 > G1 Then
          GG := G2
       Else
          GG := G1
    Else If G2 > GG then
       GG := G2

  End;

  { Print out the maximum differences }
  WriteLn(PrnFile, 'Max diff between theor. and Fourea DFT is ', DD);
  WriteLn(PrnFile, 'Max diff between orig. and inverse is ', GG)

End { of FFTTest }.
```

End Listing Two

(Listing three begins on next page)

Listing Three

```
Procedure Fourea(     numPoints : Integer;
                      whichWay  : Direction;
                 Var f          : CmplxArray);

{*
**       Performs a Cooley-Tukey type fast Fourier transform.
**
**       f is a one dimensional complex array whose length, numPoints, is
**  a power of two. whichWay determines in which direction the transform
**  will be performed. If it is Forwurd, then isInverse is set to -1 and
**  a forward transform is carried out. If whichWay has a value of Reverse,
**  then isInverse is set to 1 and a reverse transform is calculated.
**
**       transform[j] := sum(f[i]*w^((i-1)*(j-1))), where i and j run from
**  1 to numPoints and w := exp(isInverse*2*pi*sqrt(-1)/numPoints). The
**  program also computes the inverse transform, for which the defining
**  expression is: inverse DFT := (1/numPoints)*sum(f[i]*w^((i-1)*(j-1))).
**
**       Run time is proportional to numPoints*Log2(numPoints) rather than
**  to numPoints^2 for the classical discrete Fourier transform.
**
**       This is a very short version of the FFT and is only intended for
**  demonstration purposes. Programs are available which run faster and
**  are not restricted to numbers of points that are powers of two or to one
**  dimensional arrays.
*}

Var   isInverse : Integer;


  Function PowerOfTwo(     numPoints : Integer) : Boolean;
  {*
  **  Determine if numPoints is an integer power of 2.
  *}

  Var   modulo : Integer;

  Begin { PowerOfTwo }
     PowerOfTwo := True;
     modulo := 0;
     While (modulo = 0) and (numPoints >= 2) Do Begin
           modulo := numPoints Mod 2;
           If modulo = 0 Then
              numPoints := numPoints Div 2
           Else
              PowerOfTwo := False
     End
  End { of PowerOfTwo };


  Procedure Scramble(     numPoints : Integer;
                      Var f         : CmplxArray);
  {*
  **  Put the data in bit reversed order.
  *}
 Var   i,
       j,
       m    : Integer;
       temp : Complex;
```

(Continued on page 48)

C64-FORTH/79
New and Improved for the Commodore 64

C64-Forth/79™ for the Commodore 64-$99.95

- New and improved FORTH-79 implementation with extensions.
- Extension package including lines, circles, scaling, windowing, mixed high res-character graphics and sprite graphics.
- Fully compatible floating point package including arithmetic, relational, logical and transcendental functions.
- String extensions including LEFT$, RIGHT$, and MID$.
- Full feature screen editor and macro assembler.
- Compatible with VIC peripherals including disks, data set, modem, printer and cartridge.
- Expanded 167 page manual with examples and application screens.
- "SAVE TURNKEY" normally allows application program distribution without licensing or royalties.

(Commodore 64 is a trademark of Commodore)

TO ORDER
- Disk only.
- Check, money order, bank card, COD's add $1.65
- Add $4.00 postage and handling in USA and Canada
- Mass. orders add 5% sales tax
- Foreign orders add 20% shipping and handling
- Dealer inquiries welcome

PERFORMANCE MICRO PRODUCTS

MasterCard 770 Dedham Street
Canton, MA 02021
(617) 828-1209 VISA

Circle **no. 47** on reader service card.

Thanks to YOU . . . . We're Growing . . . .
with YOU and your Computer . . .

LEO ELECTRONICS, INC.
P.O. Box 11307
Torrance, CA. 90510-1307
Tel: 213/212-6133 800/421-9565
TLX: 291 985 LEO UR

We Offer . . . PRICE . . . QUALITY . . . PERSONAL SERVICE

64K UPGRADE

9 Bank (IBM PC)		$43.65	(150ns)
		$41.85	(200ns)
4164	(150ns)	$4.85 ea.	
	(200ns)	$4.65 ea.	
8 Bank (other PC)		$38.80	(150ns)
		$37.20	(200ns)
4164	(150ns)	$4.85 ea.	
	(200ns)	$4.65 ea.	

256K "Mother-Saver" Upgrade

256K - (150ns) $36.00 ea

6116P-3 — $4.40		2732 —	$3.95
2716 — $3.20		2764 —	$7.00
TMS-2716 — $4.95		27128 —	$24.00

We accept checks, Visa, Mastercard or Purchase Orders from qualified firms and institutions. U.S. Funds only. Call for C.O.D. California residents add 6½% tax. Shipping is UPS. Add $2.00 for ground and $5.00 for air. All major manufacturers. All parts 100% guaranteed. Pricing subject to change without notice.

Circle **no. 41** on reader service card.

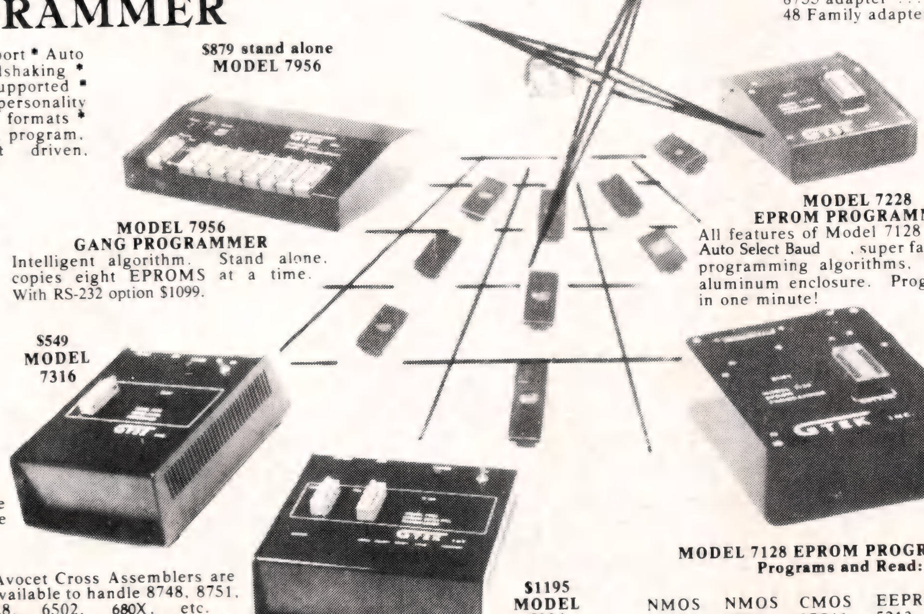
GTEK INC. DEVELOPMENT HARDWARE/SOFTWARE
HIGH PERFORMANCE/ COST RATIO
(601) 467-8048

Model 7128-L1,L2,L2A	$239.00
Model 7128-24	$329.00
DR8 or DR5	$ 30.00
DR8PGX or DR5PGX	$ 75.00
Cross Assemblers	$200.00
XASM (for MSDOS)	$250.00
U/V Eraser DE-4	$ 78.00
RS232 Cables	$ 30.00
8751 adapter	$174.00
8755 adapter	$135.00
48 Family adapter	$ 98.00

EPROM PROGRAMMER

Compatible w/all Rs 232 serial interface port * Auto select baud rate * With or without handshaking * Bidirectional Xon/Xoff and CTS/DTR supported * Read pin compatible ROMS * No personality modules * Intel, Motorola, MCS86, Hex formats * Split facility for 16 bit data paths * Read, program, formatted list commands * Interrupt driven, program and verify real time while sending data * Program single byte, block, or whole EPROM * Intelligent diagnostics discern bad and erasable EPROM * Verify erasure and compare commands * Busy light * Complete w/Textool zero insertion force socket and integral 120 VAC power (240 VAC/50Hz available)

DR Utility Package allows communication with 7128, 7228, and 7956 programmers from the CP/M command line. Source Code is provided. PGX utility package allows the same thing, but will also allow you to specify a range of addresses to send to the programmer. Verify, set the Eprom type.



$879 stand alone **MODEL 7956**

MODEL 7956 GANG PROGRAMMER
Intelligent algorithm. Stand alone, copies eight EPROMS at a time. With RS-232 option $1099.

$549 **MODEL 7228**

MODEL 7228 EPROM PROGRAMMER
All features of Model 7128 plus Auto Select Baud , super fast adaptive programming algorithms, low profile aluminum enclosure. Programs 2764 in one minute!

$549 **MODEL 7316**

$429 **MODEL 7128**

$1195 **MODEL 7324**

MODEL 7316 PAL PROGRAMMER
Programs all series 20 PALS. Software included for compiling PAL source codes.

Software Available for CPM.[1] ISIS.[2] TRSDOS[3] MSDOS.[4]

1. TM of Digital Research Corp.
2. TM of Intel Corp.
3. TM of Tandy Corp.
4. TM of Microsoft.

Avocet Cross Assemblers are available to handle 8748, 8751, Z8, 6502, 680X, etc. Available for CP/M and MSDOS computers. Order by processor type and specify kind of computer.

Model DE-4 U/V Products hold 8, 28 pin parts. High quality professional construction.

MODEL 7324 PAL PROGRAMMER
Programs all series 20 & 24 PALS. Operates stand alone or via RS232.

MODEL 7128 EPROM PROGRAMMER
Programs and Read:

NMOS	NMOS	CMOS	EEPROM	MPU'S
2508	2758	27C16	5213	8748
2516	2716	27C32	5213H	8748H
2532	2732	C6716	X2816	8749H
2564	2732A	27C54	48016	8741
68766	2764		12816A	8742H
68764	27128			8741H
8755	27256			8751
5133				

Post Office Box 289
Waveland, Mississippi 39576
[601]-467-8048

Circle **no. 29** on reader service card.

Listing Three

```
 Begin { Scramble }
    j := 1;
    For i := 1 to numPoints Do Begin
         If i < j Then Begin
            temp := f[j];
            f[j] := f[i];
            f[i] := temp
         End;
         m := numPoints Div 2;
         If m < j Then
            While m < j Do Begin
                   j := j - m;
                   m := (m + 1) Div 2
            End;
         j := j + m
    End
End { of Scramble };


Procedure Butterflies(     numPoints,
                           isInverse : Integer;
                       Var f         : CmplxArray);
{*
**  Calculate the butterflies for the bit reversed data of an FFT.
**  Normalize if a reverse FFT is performed.
*}

Const pi    = 3.1415927;

Var   mMax,
      step,
      index,
      m,
      i,
      j     : Integer;
      theta : Real;
      w,
      cn,
      temp  : Complex;

Begin { Butterflies }
   mMax := 1;
   While numPoints > mMax Do Begin
         step := 2*mMax;
         For m := 1 to mMax Do Begin
             theta := pi*(isInverse*(m - 1))/mMax;
             w.re := Cos(theta);  w.im := Sin(theta);
             For i := 1 to ((numPoints - m) Div step) + 1 Do Begin
                 index := m + ((i - 1)*step);
                 j := index + mMax;
                 CMult(temp, w, f[j]);
                 CSub(f[j], f[index], temp);
                 CAdd(f[index], f[index], temp)
             End
         End;
         mMax := step
   End;
   If isInverse = 1 Then Begin
      cn.re := numPoints;  cn.im := 0.0;
      For i := 1 to numPoints Do
```

```
            CDiv(f[i], f[i], cn)
        End
    End { of Butterflies };

Begin { Fourea }
    WriteLn('Fourea');
    If PowerOfTwo(numPoints) Then Begin
        If whichWay = Forwurd Then
            isInverse := -1
        Else
            isInverse := 1;
        Scramble(numPoints, f);
        Butterflies(numPoints, isInverse, f)
    End
    Else Begin
        WriteLn('The number of points in the array was not a power of two.');
        WriteLn('No transformation performed.')
    End
End { of Fourea };
```

"Truncated Output from the Program"

```
Complex input sequence:

( 1)      1.00000       0.00000      ( 2)    9.00000E-1    3.00000E-1
( 3)    7.20000E-1    5.40000E-1     ( 4)    4.86000E-1    7.02000E-1
            .             .                      .             .
            .             .                      .             .
            .             .                      .             .
(61)    3.80692E-2    1.86474E-2     (62)    2.86681E-2    2.82034E-2
(63)    1.73402E-2    3.39835E-2     (64)    5.41117E-3    3.57872E-2

Theoretical discrete Fourier transform:

( 1)      1.10736       2.98377      ( 2)      1.65441       4.19275
( 3)      3.60049       6.69406      ( 4)    1.58381E1       7.26182
            .             .                      .             .
            .             .                      .             .
            .             .                      .             .
(61)    6.55765E-1      1.31934      (62)    6.98674E-1      1.54663
(63)    7.65931E-1      1.85433      (64)    8.81321E-1      2.29593

"Fourea" generated discrete Fourier transform:

( 1)      1.10736       2.98377      ( 2)      1.65441       4.19275
( 3)      3.60051       6.69405      ( 4)    1.58380E1       7.26169
            .             .                      .             .
            .             .                      .             .
            .             .                      .             .
(61)    6.55781E-1      1.31934      (62)    6.98693E-1      1.54663
(63)    7.65960E-1      1.85433      (64)    8.81363E-1      2.29592

"Fourea" generated inverse descrete Fourier transform:

( 1)    9.99996E-1    1.02073E-6     ( 2)    8.99998E-1    3.00002E-1
( 3)    7.19999E-1    5.40002E-1     ( 4)    4.85999E-1    7.02002E-1
            .             .                      .             .
            .             .                      .             .
            .             .                      .             .
(61)    3.80697E-2    1.86473E-2     (62)    2.86679E-2    2.82035E-2
(63)    1.73393E-2    3.39839E-2     (64)    5.40911E-3    3.57877E-2

Max diff between theor. and Fourea DFT is  1.24931E-4
Max diff between orig. and inverse is  3.81470E-6
```

End Listings

GREP.C

A Unix-Like, Generalized, Regular Expression Parser in C

by Allen Holub

"The power of grep lies in its use of regular expressions as pattern templates rather than explicit strings."

Grep is the Unix pattern finder: it goes into a file or group of files and finds text patterns matching a symbolic regular expression. Grep is surprisingly useful. With it you can find a subroutine lost in one of the 50 modules making up the giant program that you're working on. You can find the misspelled name that your linker says is an undeclared function. Grep can number all the lines in a file or list all procedure declarations in a C program, as well as perform any of innumerable other things.

A good example of grep's utility is the history of this version, grep.c. I started out wanting to expand the pattern-searching capabilities of Ed Reams' editor, RED (*DDJ* No. 81). I wanted to add a pattern-searching capability similar to that of the Berkeley Unix editor, vi. So I set about converting into C the pattern-matching algorithms in *Software Tools in Pascal* by Kernighan and Plauger (Addison-Wesley, 1981, Ch. 5).

Along the way I ran into difficulties. My version of RED was written in BDS C, which has a nonstandard I/O library. I wanted to translate the editor over to standard C so I could port it between different compilers and different machines. To do the translation, I needed to find all calls to the nonstandard library routines. So I turned my pattern matcher into a real program, linked the BDS version of RED without linking the library modules (to get a list of the library routines that RED used), then used grep to search all the modules of RED for procedure calls to the library functions.

The program presented in this article is most of the Unix grep. The only omissions are those command line switches that are Unix dependent, the −x switch (which performs an exact line match), and the +, ?, and ( ) regular expression operators (which are not essential).

The power of grep lies in its use of regular expressions as pattern templates rather than explicit strings. For example, the grep command line:

```
grep ^[a − z][a − z]*[\s\t]*.*([^;]*)[^;]*$
     mod1.c mod2.c mod3.c
```

creates a cross reference of a large C program. The three files mod1.c, mod2.c, and mod3.c are searched. Grep's output will show all subroutine declarations along with the name of the file in which the subroutine is declared. The regular expression is interpreted as follows: beginning of line (^), followed by one or more occurrences of any character in the range a to z ([a − z][a − z]*), followed by either a space or a tab repeated zero or more times ([\s\t]*), followed by any character repeated zero or more times (.*), followed by an open parenthesis (( ), followed by any character except a semicolon repeated zero or more times ([^;]*), followed by a close parenthesis ( )), followed by any character except a semicolon repeated zero or more times ([^;]*), followed by end of line ($).

Allen Holub, Software Engineering Consultants, P.O. Box 5679, Berkeley, CA 94705.

You could also use grep to get a count of the number of procedures found and either have the matching lines printed out with line numbers or have all the lines not matching the pattern printed by including various command line switches in the program invocation.

Regular Expressions

Regular expressions are a way of representing text patterns in a symbolic shorthand. The * and ? that CP/M uses are examples of a crude regular expression syntax. The symbols grep uses to define regular expressions fall into five categories:

- Symbols that match a specific character
- Symbols that match any character
- Symbols that match a character's position on the line
- Symbols (called "character classes") that match any of a set of characters or anything except a set of characters
- Symbols that let you match the previous symbol any number of times (called "closure")

The rules for constructing regular expressions are given on the excerpted manual page (Figure 1, page 53). Some examples follow.

a.d
matches any word containing an "a," followed by any character, followed by a "d." This expression will match the substrings "and" in "*and*" and "ard" in "a*ard*vark"; this means that grep will print any *line* containing either word, along with any other match.

^a.d
will match the same strings but only if they occur at the beginning of the line. No characters, including spaces and tabs, are allowed in front of the "a."

a.d$
will match the same strings if they occur at the end of the line. No character, including spaces and tabs, can follow the "d."

^$
will match a beginning of line, followed by an end of line; this means that it will match all lines containing nothing but a newline character.

an*d
will match any word containing an "a," followed by an "n" repeated zero or more times. This expression will match "add" as well as "and."

.*
will match any character repeated any number of times. This expression will always succeed. For example, an invocation of grep with the line

grep –n .* (filename)
will output every line in the file preceded by its line number.

aa*
will match one (rather than zero) or more occurrences of the letter "a." For example,

[abc]
defines a "character class." A character class matches any one of the characters surrounded by the square brackets. This particular character class will match an "a," "b," or "c" in the corresponding position on the line.

who[ms]e
matches "who," followed immediately by either an "m" or an "s," followed by an "e." That is, the words "whomever" and "whose" will be matched, but the word "whole" will not. Character class definitions may be abbreviated by using a dash. For example,

[a – k]
will be treated as if you had said [abcdefghijk]. Similarly,

[0 – 9A – Fa – f]
will be expanded to [0123456789ABCDEFabcdef]. This last character class will match any single hexadecimal digit. That is, any digit or any letter between "a" and "f" will be matched. Note that you have to say "A – Fa – f" to match both upper and lower case.

0x[0 – 9a – fA – F][0 – 9a – fA – f]*
will match all lines in a C program that contain hexadecimal numbers. This regular expression matches the characters 0x, followed by any of the characters 0123456789abc defABCDEF repeated one or more times. A "negative character class" (one that matches any character except those listed) may be defined by using ^ as the first character following the [. For example,

f[^o]
will find all occurrences of an "f" not followed by an "o." Be careful here with patterns at the end of line. Although [^o] matches any character that is not an "o," a lone "f" at the end of line will not match the pattern because the end of line is not a character – it is a position. f[^o]*$ or f$|f[^o] will find an "f" at the end of line.

[a – zA – Z]f\s|[A – Z][A – Z]*\s
will match all lines containing words ending in "f" or all lines containing words composed only of upper-case characters. That is, if either of the regular expressions separated by the | are satisfied, a match is returned.

If you need to match a character that is used as a symbol in the regular expression, precede it with a backslash (\). For example, * will match an asterisk, and \\ will match the backslash itself. Certain escape sequences (as these backslash sequences are called) are predefined; in particular, \s matches a space and \t matches a tab (control-I).

These are needed because of the irregularities of certain compilers and operating systems. A space in the command line will make many command-line interpreters break up the expression into two arguments. A tab in the command line will confuse CP/M utterly: it won't execute your program at all. Other escape sequences are defined on the manual page (Figure 1 below).

Technical Description

The routines in tools.c differ from those in *Software Tools* in three important ways. First, the routines were translated into C. Second, all references to array indexes were replaced with pointers in the interest of increased execution speed. Finally, the data structure used for the pattern template was changed significantly.

The reasons for this last change are somewhat complex. Grep breaks up the input expression into a pattern template, where each element of the template represents a single logical portion of the expression. For example, the expression

[a – z]x.*

requires four elements in the pattern template. One element is required for the character class ("[a – z]"), one element for the literal character match ("x"), one element to match any character (the ".") and one element for the closure (the "*"). Processing the expression is much easier once it has been functionally divided in this way.

Kernighan and Plauger use a single ASCII string as their pattern template, and this data structure causes several problems. Varying numbers of characters are required to represent different types of elements in the template. To advance through the template, you need a subroutine that analyzes the current element and then advances the appropriate number of characters; this subroutine adds unnecessary overhead to the pattern-recognizing parts of grep. By replacing the ASCII string with a linked list of structures (which is how the templates are represented in my version of grep), you can advance to the next pattern with a single assignment operation.

Grep can be broken up into three distinct parts:

(1) Get the regular expression(s), the file list, and any switches from the command line.
(2) Translate the expression(s) into a pattern template.
(3) Go through the input files one line at a time, calling the routine matchs( ), and produce the appropriate output on finding a match.

Getting the Expressions

Grep is divided into two main modules: grep.c and tools.c. Grep.c does all of the I/O and tools.c contains the pattern-matching routines. Grep translates the pattern strings into a special template representing the pattern (more about this later), while the routine matchs( ) does the actual pattern matching; it processes all symbols except the OR (|) operator, which separates the regular expressions.

Grep creates a template for every regular expression input and organizes these templates using an array of pointers to templates (similar to argv) called exprv[ ]; a count of the number of separate expressions in the array (exprc) is also available. Grep then calls matchs( ), once for each template in exprcv[ ], before getting the next input line.

The Pattern Template

The templates are a linked list of structures called TOKENs

NAME

grep—search a file for a pattern

SYNOPSIS

grep [–options] . . . [expression] [filelist] . . .

DESCRIPTION

This program will find a string specified by a regular expression in a file or group of files. The following options are recognized:

–v All lines but those matching are printed.
–c Only a count of the matching lines is printed.
–l The names of the files with matching lines are listed (once) separated by newlines.
–n Each line is preceded by its line number in the file.
–h Do not print filename headers with output lines.
–y All characters in the file are mapped to upper case before matching. This is the default if the regular expression is given on the command line (because CP/M maps everything on the command line to upper case). Use the –f option if you need both lower and upper case.
–e <expression> Same as a simple expression argument, but useful when the expression begins with a "–."
–f <file> The regular expression is taken from the file. If several regular expressions are listed (separated by newlines or | s) then a match will be flagged if any of the regular expressions are satisfied. –e and –f are mutually exclusive. If –f is given, any regular expression on the command line is taken to be a filename.

Regular expressions are composed of the following:

A ^ matches the beginning of a line.

A $ matches the end of a line.

A \ followed by a single character matches that character. In this way a "*" will match an asterisk, a "\." matches a period, etc. The following sequences are special:

Figure 1

\b backspace (^H)
\n linefeed (^J this is not the same as $)
\r carriage return (^M)
\s space
\t tab (^I)
\\ backslash

A . matches any character.

A single character not otherwise endowed with special meaning matches that character.

A string enclosed in brackets [ ] specifies a "character class." Any single character in the string will be matched. For example "[abc]" will match an a, b, or c. Ranges of ASCII character codes may be abbreviated as in "[a–zO–9]." If the first symbol following the [is a ^ then a "negative character class" is specified. In this case, the string matches all characters except those enclosed in the brackets (i.e., [^a–z] matches everything except lower case letters). Note that a negative character class must match something, even though that something cannot be any of the characters listed. For example: "^$" is not the same as "^[^z]$." The first example will match an empty line (beginning of line followed by end of line); the second example matches a beginning of line followed by any character except a z followed by end of line. In the second example a character must be present on the line, but that character can't be a z. Note that *, ., ^, and $ are not special characters when inside a character class.

A regular expression followed by a * matches zero or more matches of the regular expression.

Two regular expressions concatenated match a match of the first followed by a match of the second.

Two regular expressions separated by a | or a newline match either a match for the first or a match for the second.

The order of precedence is [ ] then * concatenation then | then newline.

EXAMPLE:

The command line:

grep –n ^[a–z] [a–z] * [\s\t] *.* ([^;]*) [^;]*$ <file list>

creates a cross reference of a large C program. "<file list>" should be replaced with a list of the modules to be searched. Grep's output will show all subroutine declarations in all the listed files. In addition, every output line will be preceded by both the name of the file in which the line was found (this is automatic if more than one file is searched) and by the appropriate line number (the –n causes line numbers to be shown).

The regular expression is interpreted as follows: beginning of line (^) followed by one or more occurrences of any character in the range a to z ([a–z] [a–z]*), followed by either a space or a tab repeated zero or more times ([\s\t]*), followed by any character repeated zero or more times (.*), followed by a open parenthesis (( ), followed by any character except a semicolon repeated zero or more times ([^;]*), followed by a close parenthesis ( )), followed by any character except a semicolon repeated zero or more times ([^;]*), followed by end of line ($).

BUGS

All features of the unix version of grep are supported except that the –s, –x, and –b options and the meta-characters (,), + and ?.

Arguments, if present, must be grouped together in the second position on the command line. The character of the group must be a –. Unless the –f option is given, the next argument is always taken to be the expression. If –f is present then the third argument is the name of the file containing the expression.

Beware of spaces or tabs in the expression, even if your compiler supports quoted arguments. CP/M will object to ^I anywhere on the command line. Use \s for spaces and \t for tabs to be safe.

Some of the command line switches do mutually exclusive things (like –ef and –eh). If you try to trick grep into doing something it is not supposed to do, the output will be undefined.

Grep's execution speed varies as a function of the type of expression being parsed. The speed will vary as follows (listed fastest to slowest):

- Simple expressions anchored to beginning of line (^<expression>).
- Expressions matching literal strings.
- Expressions including character classes ([ ]).
- Expressions including closure (*).

Figure 1

(Figure 2, below). Matchs( ) is passed a pointer to this linked list. A string holding a regular expression is converted to a template by the procedure makepat( ). Since alloc( ) is used to allocate the (main) memory needed to contain the template, the expression can be any length, within reason. Using the routine unmakepat( ), you can return the memory used by a template to the free list. Unmakepat( ) is not used by grep, but it may be useful for other applications (such as editors).

Some of the fields in a TOKEN are not always used; although using a union would have saved a small amount of (main) memory space, this would have added additional complexity to the program as a whole. The tok field identifies the type of symbol represented by this node (closure, character class, etc.). If the node is a literal character, 1char holds the character itself, and if the node is a character class, string points at a string holding all the characters in the class. Classes defined with the dash notation (a – z) are expanded.

Note that a CLOSURE token is put into the chain in front of the node on which it operates (even though the closure symbol is put after the character in the expression itself). This transposition eliminates the need for any sort of lookahead in the searching routines. When we encounter the CLOSURE token, we know the next token should be repeated zero or more times. If the CLOSURE token didn't come first, we would process the character on which the CLOSURE operates as if it were a literal match; that is, we would match a single occurrence of the character. Since closure represents zero or more matches, this first match would be incorrect. So, if the CLOSURE token didn't come first, we couldn't process a token without also having to look for a CLOSURE token following it.

Matchs( )

The core of grep is the routine matchs( ) and the procedures that matchs( ) calls. This routine looks for a regular expression match in a string. It takes the string and a pointer to a pattern template as input and returns a pointer into the string upon success (or zero if no match is found). This pointer can point at either the beginning or the end of the matched string, depending on the value of the "ret_endp" parameter to matchs( ). If ret_endp is zero, a pointer to the beginning of the matched string is returned; if ret_endp is non-zero, then a pointer to the end of the matched string is returned. For example, given the string "abcdefghijklm" and the pattern a.*j, matchs( ) can return a pointer either to the "a" or to the "j." This is a useful feature in an editor.

You must be careful with the $ symbol if you want to use the pointers returned by matchs( ). Usually $ means *at* the end of line, not the end-of-line character itself; for example, if you are searching for f$, a pointer will be returned to the "f." However, if you are searching for $ itself, a pointer to the actual end-of-line character is returned. This takes care of the ^$ case. Matchs( ) must return a pointer to something, and the only character on the line is the newline character. A search for \n will always return a pointer to the newline. This last is a nonstandard feature but, again, is useful in an editor application.

Matchs( ) advances through the input string, one character at a time, until it reaches the end of the string and failure is returned. It calls amatch( ) to actually do the comparisons; when amatch( ) returns success, so does matchs( ).

Amatch( ) goes through the pattern template, one element at a time, comparing it with the text string. It advances to the next element of the template with each successful comparison, also advancing the text string as appropriate. If amatch( ) reaches the end of the template, the match is successful. Omatch( ) is called to do the simple comparisons: single characters against single elements in the pattern template.

Amatch( ) returns immediately on failure so the performance of matchs( ) is not too slow in the general case (execution time is directly proportional to the length of the input stream). The worst-case performance, however, is an exponential function of the matched string's length. Given an input string of the form

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab

along with a match string

a*c

amatch( ) will be called n times, where n is the length of the input string. Each call to amatch( ) will look at the entire input string on the order of n^2 times; this means that the total worst-case execution time is $O(n^3)$.

Most of this is the fault of closure processing, which is done by brute force. Amatch( ) first eliminates all the characters defined by the closure, scanning along the text string and calling omatch( ) until a mismatch is found. It then tries to match the rest of the template against the rest of the text string. If it fails to do so, amatch( ) goes backwards through the characters it just processed, still trying to match the trailing string against the rest of the pattern template. This is necessary because the character following the closure could have been included in the closure itself.

For example, in the pattern [a – z]*t (which matches any

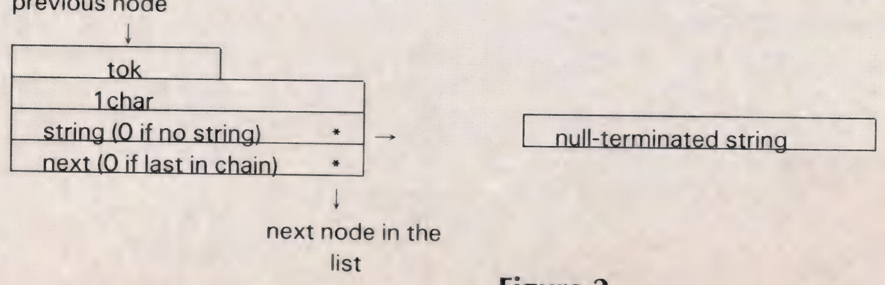


Figure 2

lower-case word ending in a "t"), the final "t" will be sucked up by the first scan since "t" is included in the character class [a – z]. Since amatch( ) has scanned too far, an attempt to match the "t" will now fail. So, it backs up a notch in the input string then tries to match the rest of the pattern template again, repeating this process recursively until it gets back to the beginning of the closure. The recursion only goes one level deep. If anyone knows a better way to do this, please tell me.

Examples

Makepat( ) takes two arguments. The first argument is an ASCII string holding the regular expression; the second argument is a character to use as a terminator in the expression string. That is, processing of the expression string will be terminated when the character specified by the second argument is encountered.

The template returned by makepat( ) when called with

```
makepat("^The qui", '\0')
```

is shown in Figure 3 (below). If we had called makepat with

```
makepat("^The quick", ' ')
```

a template like that in Figure 3 would be returned. However, this time the template would stop with the "e" LITCHAR because we passed makepat( ) a space as the input string terminator. A call to

```
makepat("a[0 – 9].*[^v]$",' \0')
```

returns a pointer to the template shown in Figure 4 (page 56).

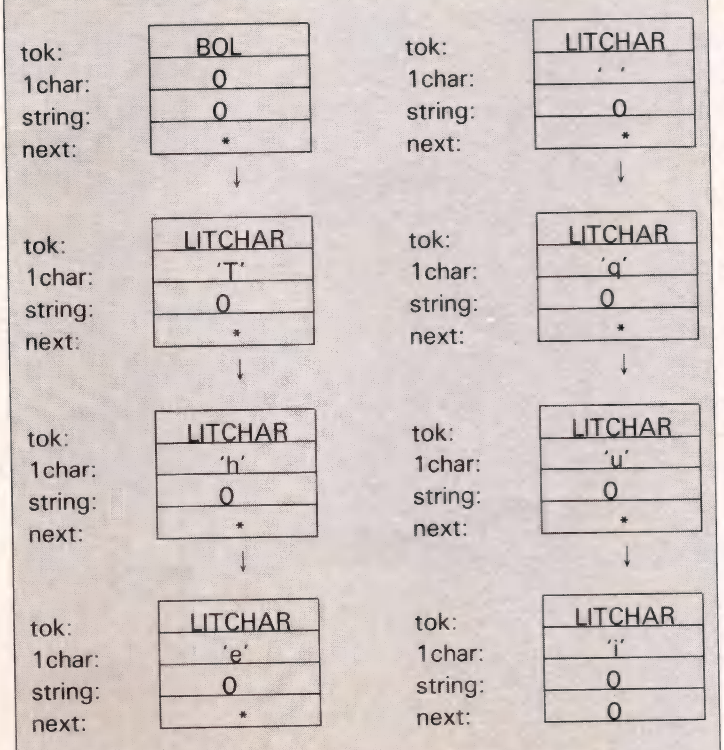


Figure 3

Matchs( ) looks for an expression on only one input line. Consequently, it has to be called several times, once for each line in the input file. Three arguments are required: the first is a pointer to the line being searched, the second is a pointer to a pattern template (returned by a previous call to makepat( ), and the third determines whether matchs( ) will return a pointer to the beginning or to the end of the matched pattern (0 for a pointer to the beginning, 1 for the end).

Consider the program fragment:

```
#include "tools.h"
TOKEN *template; char *ptr;
template = makepat("456", '\0');
ptr = matchs("1234567890", template, 0);
ptr = matchs("1234567890", template, 1);
ptr = matchs("abcdefghij", template, 1);
```

The call to makepat( ) returns a pointer to a pattern template representing the string "456." This template will be three elements long, one element for each character in the string, and all three elements will be of type LITCHAR. The first call to matchs( ) will return a pointer to the "4" (because its third parameter is zero). The second call to matchs( ) will return a pointer to the "6" (because the third parameter is 1). The third call to matchs( ) returns zero because the string "456" doesn't exist in the string "abcdefghij."

A simplistic version of grep—using only gets( ), makepat( ), and matchs( )—is shown in Figure 5 (page 57). This version prints all input lines that match a pattern found on the command line. No attempt at any sort of error checking is made in this example, so it's not a very practical program. It does illustrate how makepat( ) and matchs( ) may be used in a real program.

Debugging Aids

Four routines are included here for use in debugging. These are pr_tok( ), pr_line( ), insert( ), and delete( ); all are in tools.c.

Pr_tok( ), when passed a pointer to a linked list of TOKENs, will print out the list to stdout. You can use pr_tok( ) to monitor the progress of amatch( ) as it works and to see if the expression is translated correctly to begin with.

Pr_line( ) prints out one line of text to stdout; any non-printable characters are represented as numbers in the form:

\0x(two hex digits)

Insert( ) puts a character into a string at a place pointed to by its "str" parameter. Delete( ) takes the character out again.

Some Implementation Notes

In the course of bringing up grep on my own system, I ran into a few problems worth mentioning. First, depending on which compiler you use, getting the expression from the command line may be unexpectedly difficult. The command-line parser for Aztec C II (the compiler I used on this version) doesn't allow quoted strings; that is, an argument of the form

grep "this is a single argument" foo.bar

will be broken up into seven (rather than three) arguments by the compiler. So I modified the command-line parser in the module croot.c to accept quoted strings.

While the BDS C compiler does give you quoted strings, pitfalls exist here too. As it parses the command line, BDS strips off the quotes. Because BDS wild card expansion is done with a call to wildexp( ) inside of the program proper (instead of inside the command-line parser where it belongs), wildexp( ) can't differentiate between the quoted argument and the normal arguments: it doesn't have any quotes to work with. Consequently, it will try to expand the regular expression if the expression has an * or a ? in it.

I got around this problem in a BDS version of grep by getting the regular expression from the command line *before* calling wildexp( ). I then replaced the argv entry, which pointed at the expression, with a pointer to a null string and called wildexp( ). You could also try to call wildexp( ) from inside the command-line parser itself. Since the parser is written in assembly language and wildexp( ) is in C, I didn't try this (though it would be a permanent solution to the problem).

A similar problem can be found in microshell. Microshell supports quoted strings, but a backslash inside a quoted string is treated as a special character. You need to double the backslash to pass it through to grep (i.e., use \\s instead of \s; and \\\\ instead of \\). One saving grace is that microshell lets you pass tabs through unmolested.

One final difficulty. You may use grep as a filter if you like: if you are running microshell or some other environment that supports pipes, you may use grep as a general purpose filter, stripping out unwanted material from the input stream and passing the modified stream on to another program. The problem is what happens to end-of-line terminators on their way through the pipe.

CP/M requires a carriage return, line feed combination at the end of line. C, however, wants a single newline character (\n). Consequently, when getc( ) sees a CR, it echoes it as a CR–LF (so the screen looks nice) and then turns it into a \n; on output, putc( ) will turn the \n back into a CR–LF.

This is fine until you use the output of one program as the input of another. The next program will see the CR–LF, echo it as CR–LF–LF, and map it to a \n\n (one \n for the CR, another for the LF). The output from the second program will have a CR–LF–CR–LF at the end of every line: instant double spacing. If you go through another layer of pipe you will get CR–LF–CR–LF–CR–LF–CR–LF at the end of each line, and so on.

A solution would be to have getchar( ) work as described above and have getc( ) ignore the LF character entirely (not pass it through to the program). The BDS C compiler doesn't lend itself to this change because its I/O library has the two input routines functionally reversed (i.e., getc( ) calls getchar( ), which is backwards from Unix). You could also use the BDS version 1.5 raw I/O routines, but then your code would be even more nonstandard. Alternately, you could do all your character input from the console with direct bdos( ) calls.

Conclusion

In spite of the few implementation problems I encountered, grep remains an extremely useful program. It has saved me hours of rooting around in modular C programs looking for misspelled subroutine names. Its cross-referencing capability has also proved invaluable. When I get a new C compiler, the first thing I do is use grep to make a cross reference of the

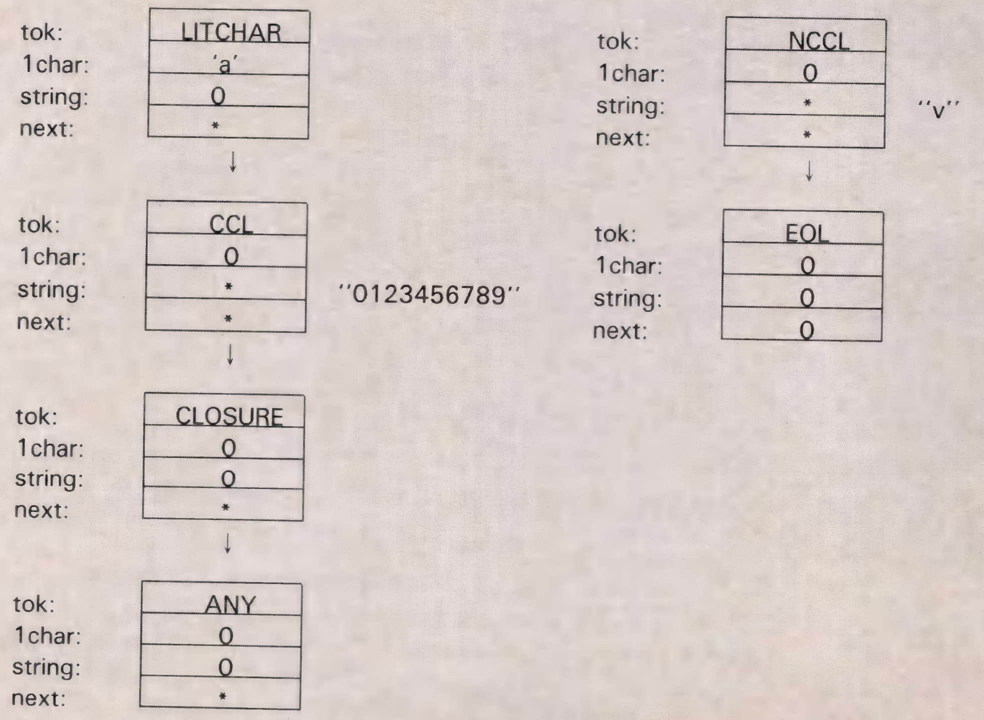


Figure 4

```
#include "stdio.h"
#include "tools.h"

main(argc, argv)
int    argc;
char  *argv[ ];
{M8•X
        TOKEN  *template;
        char      str[132];

        template = makepat( argv[1], '\0' );

        while(    gets( str ) != NULL )
                   if( matches( str, template, 0 ) )
                         printf("%s\n", str );
}
```

Figure 5

runtime library sources. Using this cross reference, it's easy to find the source code for the particular library subroutine that doesn't seem to be working correctly . The addition of matchs( ) to the RED editor has made it a much nicer editor, giving RED not only an extended search capability but also a powerful global substitution capability. Once you've used regular expressions in an editor, you won't settle for anything else. I hope that you find this program as useful as I have.

Copies of grep, along with the C source code, are available from the author at the above address for $35.00 (+ tax if a California resident). (Copies provided on a standard, IBM format, single density, 8-inch, CP/M-compatible floppy disk or on DS/DD, IBM-PC, PCDOS-Compatible, 5¼-inch disks).

DDJ

GREP.C (Listing Continued, text begins on page 50)

Listing One

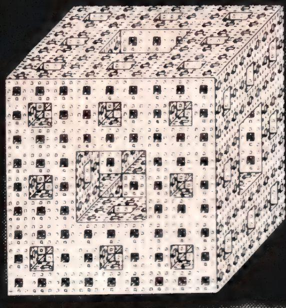
```
/*------------------------------------------------------------------
 *        TOOLS.H: Various #defines and typedefs for GREP
 *
 *                  Copyright (c) 1984 Allen Holub
 *      Copyright (c) 1984 Software Engineering Consultants
 *                         P.O. Box 5679
 *                     Berkeley, CA,  94705
 *
 *                      All rights reserved.
 *
 *      This program may be copied for personal,  non-commercial use
 *      only, provided that this copyright notice is included in all
 *      copies  and  that  this  program is not modified in any way.
 *      Copying for any other use without previously  obtaining  the
 *      written permission of the author is prohibited.
 *
 *      Machine readable versions of  this  program may be purchased
 *      for $35 from  Software  Engineering  Consultants.  Supported
 *      disk formats are CP/M 8" SS/SD and PCDOS (v2.x) 5-1/4" DS/DD.
 *
 *------------------------------------------------------------------
 */

/*
 *      #defines for non-printing ASCII characters
 */

#define NUL     0x00    /* ^@   */
#define SOH     0x01    /* ^A   */
#define STX     0x02    /* ^B   */
#define ETX     0x03    /* ^C   */
#define EOT     0x04    /* ^D   */
#define ENQ     0x05    /* ^E   */
#define ACK     0x06    /* ^F   */
#define BEL     0x07    /* ^G   */
#define BS      0x08    /* ^H   */
#define HT      0x09    /* ^I   */
#define LF      0x0a    /* ^J   */
#define NL      LF
#define VT      0x0b    /* ^K   */
#define FF      0x0c    /* ^L   */
#define CR      0x0d    /* ^M   */
```

(Continued on next page)

GREP.C (Listing Continued, text begins on page 50)
Listing One

```
#define SO      0x0e    /* ^N   */
#define SI      0x0f    /* ^O   */
#define DLE     0x10    /* ^P   */
#define DC1     0x11    /* ^Q   */
#define DC2     0x12    /* ^R   */
#define DC3     0x13    /* ^S   */
#define DC4     0x14    /* ^T   */
#define NAK     0x15    /* ^U   */
#define SYN     0x16    /* ^V   */
#define ETB     0x17    /* ^W   */
#define CAN     0x18    /* ^X   */
#define EM      0x19    /* ^Y   */
#define SUB     0x1a    /* ^Z   */
#define CPMEOF  SUB
#define ESC     0x1b    /* ^[   */
#define FS      0x1c    /* ^\   */
#define GS      0x1d    /* ^]   */
#define RS      0x1e    /* ^^   */
#define US      0x1f    /* ^_   */
#define DEL     0x7f    /* DEL  */


#define TRUE    1
#define FALSE   0


/*      Definitions of meta-characters used in pattern matching routines.
 *      LITCHAR & NCCL are only used as token identifiers; all the others
 *      are also both token identifiers and the actual symbol used in
 *      the regular expression
 */


#define BOL     '^'
#define EOL     '$'
#define ANY     '.'
#define LITCHAR 'L'
#define ESCAPE  '\\'
#define CCL     '['             /* Character class:     [...]           */
#define CCLEND  ']'
#define NEGATE  '^'
#define NCCL    '!'             /* Negative character class [^...]      */
#define CLOSURE '*'
#define OR_SYM  '|'

#define CLS_SIZE  128           /* Largest permitted size for an expanded
                                ** character class.  (Ie. the class [a-z]
                                ** will expand into 26 symbols; [a-z0-9] will
                                ** expand into 36 symbols.)
                                */

/*
 *      Tokens are used to hold pattern templates. (see makepat() in

 *      tools.h
 */

typedef struct token{
                char            tok;
                char            lchar;
                char            *string;
                struct token    *next;
              }TOKEN;

#define TOKSIZE sizeof(TOKEN)
```

(Continued on page 60)



WALTZ LISP(TM)

*The one and only **adult** Lisp system for CP/M users.*

Waltz Lisp is a very powerful and complete implementation of the Lisp programming language. It includes features previously available only in large Lisp systems. In fact, Waltz is substantially compatible with *Franz* (the Lisp running under *Unix*), and is similar to *MacLisp*. Waltz is perfect for Artificial Intelligence programming. It is also most suitable for general applications.

Much faster than other microcomputer Lisps. • Long integers (up to 611 digits). Selectable radix • True dynamic character strings. Full string operations including fast matching/extraction. • Flexibly implemented random file access. • Binary files. • Standard CP/M devices. • Access to disk directories. • Functions of type lambda (expr), nlambda (fexpr), lexpr, macro. • Splicing and non-splicing character macros. • User control over all aspects of the interpreter. • Built-in prettyprinting and formatting facilities. • Complete set of error handling and debugging functions including user programmable processing of undefined function references. • Virtual function definitions. • Optional automatic loading of initialization file. • Powerful CP/M command line parsing. • Fast sorting/merging using user defined comparison predicates. • Full suite of mapping functions, iterators, etc. • Assembly language interface. • Over 250 functions in total. • The best documentation ever produced for a micro Lisp (300+ full size pages, hundreds of illustrative examples).

Waltz Lisp requires CP/M 2.2, Z80 and 48K RAM (more recommended). All common 5'' and 8'' disk formats available.

ProCode (TM)
INTERNATIONAL
15930 SW Colony Pl.
Portland, OR 97224

Unix* Bell Laboratories.
CP/M* Digital Research Corp.

Version 4.4

(Now includes Tiny Prolog written in Waltz Lisp.)

$169*

*Manual only: $30 (refundable with order). All foreign orders: add $5 for surface mail, $20 for airmail. COD add $3. Apple CP/M and hard sector formats add $15.

Call free **1-800-LIP-4000** Dept. #11
In Oregon and outside USA call 1-503-684-3000

Circle **no. 51** on reader service card.

PROGRAMMER'S DEVELOPMENT TOOLS

IBM Personal Computer Language and Utility Specialists

LANGUAGES:

	List	Ours
C-86 Computer Innovations	$395	319
C Programming System by Mark Williams	500	459
Lattice C Compiler	500	299
Professional BASIC Morgan Computing	345	295
ADA-86 + Tools Janus	700	499

Call for Microsoft and Digital Research Products.

*** ''C'' Language Starter Kit ***

Package Consists of:	List	Ours
DeSmet C Compiler w/Debugger	$159	145
Windows For C Creative Solutions	150	119
C Programming Language book by K & R	25	20

Retail $334. Priced Separately $284

Our Special Package Price! $269

Greenleaf Utilities available for DeSmet C. Call for Details and Prices.

We have in-staff APL expertise!

**** STSC APL*Plus/PC ****

This powerful, interactive, fourth-generation language includes a tutorial, help system and useful extensions. Comes with plug-in APL character generator chip.

Retail $595 Our Normal Price $540

Special Sale Price! $469

Send for complete demonstration package **$5**

UTILITIES:

	List	Ours
C Functions Library by Greenleaf	175	159
Btrieve by SoftCraft	245	205
Communications Library by Greenleaf *New*	160	139
Trace-86 by Morgan Computing	125	115
OPT-TECH Sort *High Performance Utility*	99	87
Profiler by DWB & Associates	175	149
AKA ALIAS by Soft Shell Technology	60	57
Plink-86 *Overlay Linkage Editor*	395	315
Panel *Screen Design/Editing* by Roundhill	350	259
FirsTime *Intelligent C Text Editor* by Spruce	295	CALL
Halo *Color Graphics for Lattice, CI-86*	200	159
Windows For C by Creative Solutions	150	119

*** A SOLID GOLD VALUE ***

CodeSmith-86 Debugger
Version 1.8 by Visual Age

Retail $145, Our Normal Price $129

Special Sale Price! $109

Prices are subject to change without notice.

Account is charged when order is shipped.

Visa/MC NO EXTRA CHARGE

CALL FOR LOW PRICES

1-800-336-1166

Programmer's Connection
281 Martinel Drive
Kent, Ohio 44240
(216) 678-4301 (In Ohio)

''Programmers Serving Programmers''

Circle **no. 52** on reader service card.

Re-ink any fabric ribbon for less than 5¢. Extremely simple operation. We have a MAC INKER for any printer. Lubricant ink safe for dot matrix printheads. Multi-colored inks, uninked cartridges available. Ask for brochure. Thousands of satisfied customers.

$54.95 +

Mac Switch lets you share your computer with any two peripherals (serial or parallel). Ideal for word processors—never type an address twice. Ask us for brochure with tips on how to share two peripherals with MAC SWITCH. Total satisfaction or full refund.

$99.00

Order Toll Free 1-800-547-3303

Computer Friends
6415 SW Canyon Court
Suite #10
Portland, Oregon 97225
(503) 297-2321

Mac Inker & MacSwitch

Circle **no. 14** on reader service card.

Listing One

```
/*
 *      An absolute maximum for strings.
 */

#define MAXSTR          132             /* Maximum number of characters in
                                         *      a line.
                                         */

extern  char    *matchs();
extern  int     amatch();
extern  char    *in_string();
extern  TOKEN   *getpat();
extern  int     esc();
extern  int     dodash();
extern  TOKEN   *makepat();
extern  int     unmakepat();
extern  int     insert();
extern  int     delete();
extern  int     isalphanum();
extern  int     stoupper();
extern  int     pr_tok();
extern  int     pr_line();
extern  int     max();
```

End Listing One

Listing Two

```
/*-----------------------------------------------------------------
 *        TOOLS.C: The expression parser used by grep.
 *
 *                  Copyright (c) 1984 Allen Holub
 *      Copyright (c) 1984 Software Engineering Consultants
 *                          P.O. Box 5679
 *                      Berkeley, CA,  94705
 *
 *                      All rights reserved.
 *
 *      This program may be copied for personal,  non-commercial use
 *      only, provided that this copyright notice is included in all
 *      copies  and  that  this  program is not modified in any way.
 *      Copying for any other use without previously  obtaining  the
 *      written permission of the author is prohibited.
 *
 *      Machine readable versions of  this  program may be purchased
 *      for $35 from  Software  Engineering  Consultants.  Supported
 *      disk formats are CP/M 8" SS/SD and PCDOS (v2.x) 5-1/4" DS/DD.
 *
 *-----------------------------------------------------------------
 */

#include "a:stdio.h"
#include "b:tools.h"

/*
 *      This module contains the various routines needed by grep
 *      to match regular expressions. Routines are ordered
 *      alphabeticaly.
 */

int     amatch( lin, pat, boln )
char    *lin, *boln;
TOKEN   *pat;
{
```

```
        /*    Scans through the pattern template looking for a match
         * with lin. Each element of lin is compared with the template
         * until either a mis-match is found or the end of the template
         * is reached. In the former case a 0 is returned; in the latter,
         * a pointer into lin (pointing to the last character in the
         * matched pattern) is returned.
         *
         *      "lin"   is a pointer to the line being searched.
         *      "pat"   is a pointer to a template made by makepat().
         *      "boln"  is a pointer into "lin" which points at the
         *                      character at the beginning of line.
         */

register char   *bocl, *rval, *strstart;


if (pat == 0)
        return (0);

strstart = lin;

while ( pat )
{
        if (pat->tok == CLOSURE  &&   pat->next)
        {
                /*
                 *      Process a closure:
                 *      First skip over the closure token to the
                 *      object to be repeated. This object can be
                 *      a character class.
                 */

                pat = pat->next;

                /*      Now match as many occurrences of the
                 *      closure pattern as possible.
                 */

                bocl = lin;

                while ( *lin  &&  omatch(&lin, pat)  )
                        ;


                /*      'Lin' now points to the character that made
                 *      made us fail. Now go on to process the
                 *      rest of the string. A problem here is
                 *      a character following the closure which
                 *      could have been in the closure.
                 *      For example, in the pattern "[a-z]*t" (which
                 *      matches any lower-case word ending in a t),
                 *      the final 't' will be sucked up in the while
                 *      loop. So, if the match fails, we back up a
                 *      notch and try to match the rest of the
                 *      string again, repeating this process
                 *      recursively until we get back to the
                 *      beginning of the closure. The recursion
                 *      goes, at most, two levels deep.
                 */


                if (pat = pat->next)
                {
                        while ( bocl <= lin )
                        {
                                if (rval = amatch(lin, pat, boln) )
                                {
                                        /* success */
                                        return(rval);
                                }
                                else
                                        --lin;
                        }
                        return (0);     /* match failed */
```

(Continued on next page)

```
                        }
                }
                else if ( omatch(&lin, pat, boln) )
                {
                        pat = pat->next;
                }
                else
                {
                        return (0);
                }
        }

        /*
         *      Note that omatch() advances lin to point at the next
         *      character to be matched. Consequently, when we reach
         *      the end of the template, lin will be pointing at the
         *      character following the last character matched.
         *      The exceptions are templates containing only a
         *      BOLN or EOLN token. In these cases omatch doesn't
         *      advance.
         *
         *      So, decrement lin to make it point at the end of the
         *      matched string. Then, check to make sure that we haven't
         *      decremented past the beginning of the string.
         *
         *      A philosophical point should be mentioned here. Is $
         *      a position or a character? (Ie. does $ mean the EOL
         *      character itself or does it mean the character at the end of
         *      the line.) I decided here to make it mean the former, in
         *      order to make the behavior of amatch() consistent. If you
         *      give amatch the pattern ^$ (match all lines consisting only
         *      of an end of line) then, since something has to be returned,
         *      a pointer to the end of line character itself is returned.
         *
         *      One final point. If you use a macro instead of a real
         *      subroutine to define max(), then take the --lin out of
         *      the macro call to avoid side-effects (lin being decremented
         *      twice).
         */

        return ( max(strstart , --lin) );
}

/* ---------------------------------------------------------------------- */

delete( ch, str )
int             ch;
register char   *str;
{
        /*      Delete the first occurrence of character from string
         *      moving everything else over a notch to fill the hole.
         */

        ch &= 0xff;

        while ( *str && *str != ch)
                str++;

        while ( *str )
        {
                *str = *(str+1);
                str++;
        }
}

/* ---------------------------------------------------------------------- */
```

(Continued on page 64)

GREP.C (Listing Continued, text begins on page 50)

(Listing Continued, text begins on page 50)

Listing Two

```
int     dodash(delim, src, dest, maxccl)

int     delim, maxccl;
char    **src, *dest;
{
        /*      Expand the set pointed to by *src into dest.
         *      Stop at delim. Return 0 on error or
         *      size of character class on success. Update *src to point
         *      at delim. A set can have one element {x} or several
         *      elements ( {abcdefghijklmnopqrstuvwxyz} and {a-z}
         *      are equivalent ). Note that the dash notation is expanded
         *      as sequential numbers. This means (since we are using the
         *      ASCII character set) that a-Z will contain the entire alphabet
         *      plus the symbols:  [\]^_`  . The maximum number of characters
         *      in a character class is defined by maxccl.
         */

        register char   *dstart;
        register int    k, at_begin;
        char            *sptr;

        dstart = dest;
        sptr = *src;
        at_begin = 1;

        while ( *sptr  &&  (*sptr != delim)   &&   (dstart-dest < maxccl)  )
        {
                if ( *sptr == ESCAPE )
                {
                        *dest++ = esc(&sptr);
                        sptr++;
                }

                else if ( *sptr != '-')
                        *dest++ = *sptr++;

                else if ( at_begin ||  *(sptr+1) == delim )
                        *dest++ = '-';                          /* literal '-' */

                else if ( *(sptr -1 ) <= *(sptr+1) )
                {
                        sptr++;

                        for( k= *(sptr-2) ; ++k <= *sptr ;)
                                *dest++ = k;

                        sptr++;
                }
                else
                {
                        return(0);
                }

                at_begin = 0;
        }

        *dest++ = '\000' ;
        *src = sptr;

        return (dest - dstart);
}

/* ------------------------------------------------------------------------- */

int     esc(s)
char    **s;
```

```
{
        register int    rval;

        /* Map escape sequences into their equivalent symbols. Returns the
         * Correct ASCII character. If no escape prefix is present
         * then s is untouched and  *s is returned, otherwise **s
         * is advanced to point at the escaped character and the
         * translated character is returned.
         */

        if ( **s != ESCAPE )
        {
                rval = **s;
        }
        else
        {
                (*s)++;

                switch( toupper(**s) )
                {
                case '\000':    rval = ESCAPE;          break;
                case 'S':       rval = ' '  ;           break;
                case 'N':       rval = '\n' ;           break;
                case 'T':       rval = '\t' ;           break;
                case 'B':       rval = '\b' ;           break;
                case 'R':       rval = '\r' ;           break;
                default:        rval = **s  ;           break;
                }
        }

        return (rval);
}

/* ------------------------------------------------------------------------ */

TOKEN   *getpat( arg )
char    *arg;
{
        /*      Translate arg into a TOKEN string
         */

        return ( makepat(arg, '\000' ) );
}

/* ------------------------------------------------------------------------ */

insert( ch, str )
int             ch;
register char   *str;
{
        /*      Insert ch into str at the place pointed to by str. Move
         *      everything else over a notch
         */

        register char   *bp;

        bp = str;

        while (*str)                    /* Find the end of string        */
                str++;
        do                              /* Move the tail over one notch */
        {
                *(str+1) = *str;
                str--;

        } while (str >= bp);

        *bp = ch;                       /* Put the char in the hole.     */
}

/* ------------------------------------------------------------------------ */

char    *in_string( delim, str )
register int    delim;
```

(Continued on next page)

Listing Two

```
register char   *str;
{
        /*
         *  Return a pointer to delim if it is in the string, 0 if it is not.
         */

        delim &= 0x7f;

        while (*str  &&  *str != delim)
                str++;

        return ( *str ? str : 0 );
}

/* ------------------------------------------------------------------------ */

int isalphanum(c)
int     c;
{

        /*   Return true if c is an alphabetic character or digit,
         *      false otherwise.
         */

        return( ('a' <= c  &&  c <= 'z') ||
                ('A' <= c  &&  c <= 'Z') ||
                ('0' <= c  &&  c <= '9')
              );
}

/*------------------------------------------------------------------------*/

TOKEN   *makepat(arg, delim)
char    *arg;
int     delim;
{
        /*      Make a pattern template from the string pointed to by arg.
         *      Stop when delim or '\000' or '\n' is found in arg.
         *      Return a pointer to the pattern template.
         *
         *      The pattern templates used here are somewhat different
         *      than those used in the book; each token is a structure
         *      of the form TOKEN (see tools.h). A token consists of
         *      an identifier, a pointer to a string, a literal
         *      character and a pointer to another token. This last is 0 if
         *      there is no subsequent token.
         *
         *      The one strangeness here is caused (again) by CLOSURE which
         *      has to be put in front of the previous token. To make this
         *      insertion a little easier, the 'next' field of the last
         *      token in the chain (the one pointed to by 'tail') is made
         *      to point at the previous node. When we are finished,
         *      tail->next is set to 0.
         */

        TOKEN   *head, *tail;
        TOKEN   *ntok;
        char    buf[CLS_SIZE];
        int     error;

        /*      Check for characters that aren't legal at the beginning
         *      of a template.
         */

        if (*arg=='\0' ||  *arg==delim  ||  *arg=='\n' || *arg==CLOSURE)
                return(0);
```

(Continued on page 68)

Circle **no. 22** on reader service card.

Circle **no. 11** on reader service card.

Listing Two

```
	error = 0;
	head = 0;
	tail = 0;

	while ( *arg  &&  *arg != delim  &&   *arg != '\n'  &&  !error)
	{
		ntok = alloc( TOKSIZE );
		ntok->string = &(ntok->lchar);
		ntok->lchar = '\000';
		ntok->next = 0;

		switch(*arg)
		{
		case ANY:
			ntok->tok = ANY;
			break;

		case BOL:

			if (head==0)	/* then this is the first symbol */

				ntok->tok = BOL;
			else
				error = 1;
			break;

		case EOL:

			if ( *(arg+1) == delim  || *(arg+1) == '\000'
						|| *(arg+1) == '\n'  )
				ntok->tok = EOL;
			else
				error = 1;
			break;

		case CCL:

			if (*(arg+1) == NEGATE)
			{
				ntok->tok = NCCL;
				arg += 2;
			}
			else
			{
				ntok->tok = CCL;
				arg++;
			}

			error = dodash(CCLEND, &arg, buf, CLS_SIZE) ;
			if (error != 0)
			{
				ntok->string = alloc( error );
				strcpy( ntok->string, buf );
				error = 0;
			}

			break;

		case CLOSURE:

			if ( head != 0 )
			{
				switch ( tail->tok )
				{
				case BOL:
				case EOL:
```

```
                                case CLOSURE:
                                        return(0);

                                default:
                                        ntok->tok = CLOSURE;
                                }
                        }
                        break;

                default:
                        ntok->tok = LITCHAR;
                        ntok->lchar = esc(&arg);
                }

                if( error  ||  ntok == 0 )
                {
                        unmakepat(head);
                        return (0);
                }

                else  if (head == 0)
                {
                        /*  This is the first node in the chain.
                         */

                        ntok->next = 0;
                        head = tail = ntok;
                }

                else if (ntok->tok != CLOSURE)

                {
                        /* Insert at end of list (after tail) */

                        tail->next = ntok;
                        ntok->next = tail;
                        tail = ntok;
                }

                else if(head != tail)
                {
                        /* More than one node in the chain. Insert the
                         * CLOSURE node immediately in front of tail.
                         */

                        (tail->next)->next = ntok;
                        ntok->next = tail;
                }

                else
                {
                        /* Only one node in the chain, Insert the CLOSURE
                         * node at the head of the linked list.
                         */

                        ntok->next = head;
                        tail->next = ntok;
                        head = ntok;
                }

                arg++;
        }

        tail->next = 0;
        return (head);
}

/* ---------------------------------------------------------------------- */

char    *matchs(line, pat, ret_endp)
char    *line;
TOKEN   *pat;
```

(Continued on next page)

Listing Two

```
int     ret_endp;
{
        /*
         *      Compares line and pattern. Line is a character string while
         *      pat is a pattern template made by getpat().
         *      Returns:
         *              1. A zero if no match was found.
         *              2. A pointer the last character
         *                 satisfying the match if ret_endp is non-zero.
         *              3. A pointer to the beginning of the matched string
         *                 if ret_endp is 0;
         *
         *      For example:
         *
         *              matchs ("1234567890", getpat("4[0-9]*7"), 0);
         *
         *      will return a pointer to the '4', while
         *
         *              matchs ("1234567890", getpat("4[0-9]*7"), 1);
         *
         *      will return a pointer to the '7'.
         */

        char    *rval, *bptr;

        bptr = line;

        while (*line)
        {
                if ( (rval = amatch(line, pat, bptr)) == 0 )
                {
                        line++;
                }
                else
                {
                        rval = ret_endp ? rval : line ;
                        break;
                }
        }

        return (rval);
}

/* ------------------------------------------------------------------------ */

stoupper(str)
char    *str;
{
        /*
         *      Map the entire string pointed to by str to upper case
         *      Return str.
         */

        char    *rval;


        rval = str;

        while (*str)
        {
                if ( 'a' <= *str  &&  *str <= 'z' )
                        *str -= ('a' - 'A');

                str++;
        }
```

```
        return(rval);
}

/* ---------------------------------------------------------------- */

int     max(x,y)
int     x,y;
{
        return ( (x>y) ?  x  :  y );
}

/* ---------------------------------------------------------------- */

int     omatch (linp, pat, boln)
char    **linp, *boln;
TOKEN   *pat;
{
        /*      Match one pattern element, pointed at by pat, with the
         *      character at **linp. Return non-zero on match.
         *      Otherwise, return 0. *Linp is advanced to skip over the
         *      matched character; it is not advanced on failure. The
         *      amount of the advance is 0 for patterns that match null
         *      strings, 1 otherwise. "boln" should point at the position
         *      that will match a BOL token.
         */



        register int    advance;

        advance = -1;

        if ( **linp )
        {
                switch ( pat->tok )
                {
                case LITCHAR:
                        if ( **linp == pat->lchar )
                                advance = 1;
                        break;


                case BOL:
                        if ( *linp == boln )
                                advance = 0;
                        break;

                case ANY:
                        if ( **linp != '\n' )
                                advance = 1;
                        break;

                case EOL:
                        if ( **linp == '\n' )
                                advance = 0;
                        break;

                case CCL:
                        if( in_string (**linp, pat->string) )
                                advance = 1;
                        break;

                case NCCL:
                        if ( ! in_string (**linp, pat->string) )
                                advance = 1;
                        break;

                default:
                        printf("omatch: can't happen\n");
                }
        }
```

(Continued on next page)

Listing Two

```
        if (advance >= 0)
                *linp += advance;

        return( ++advance );
}
/* ------------------------------------------------------------------------ */

pr_line(ln)
register char   *ln;
{
        /*      Print out ln, if a non-printing character is found, print
         *      out its numerical value in the form "\0x<hex number>".
         *      Again, this is a debugging aid. It lets you see what's
         *      really on the line.
         */

        for ( ; *ln ; ln++ )


        {
                if ( (' ' <= *ln) && (*ln <= '~') )
                        putchar(*ln);
                else
                {
                        printf("\\0x%02x", *ln);

                        if (*ln == '\n')
                                putchar('\n');
                }
        }
}

/* ------------------------------------------------------------------------ */

pr_tok(head)
TOKEN   *head;
{
        register char   *str;

        /*      Print out the pattern template (linked list of TOKENs)
         *      pointed to by head. This is a useful debugging aid. Note
         *      that pr_tok() just scans along the linked list, terminating
         *      on a null pointer; so, you can't use pr_tok from inside
         *      makepat() because tail->next points to the previous
         *      node instead of being null.
         */

        for (; head ; head = head->next )
        {
                switch (head->tok)
                {
                case BOL:
                        str = "BOL";
                        break;

                case EOL:
                        str = "EOL";
                        break;

                case ANY:
                        str = "ANY";
                        break;

                case LITCHAR:
                        str = "LITCHAR";
                        break;
```

```
            case ESCAPE:
                    str = "ESCAPE";
                    break;

            case CCL:
                    str = "CCL";
                    break;

            case CCLEND:
                    str = "CCLEND";
                    break;

            case NEGATE:
                    str = "NEGATE";
                    break;

            case NCCL:
                    str = "NCCL";
                    break;

            case CLOSURE:
                    str = "CLOSURE";
                    break;

            default:
                    str = "**** unknown ****";
            }

            printf("%-8s at: 0x%x, ", str, head);

            if (head->tok == CCL || head->tok == NCCL)
                    printf ("string =[%s]=, ", head->string );

            else if (head->tok == LITCHAR)
                    printf("lchar = %c, ", head->lchar);

            printf("next = 0x%x\n", head->next);
        }

        putchar('\n');
}

/* ----------------------------------------------------------------- */

unmakepat(head)
TOKEN   *head;
{
        /*      Free up the memory used for the token string    */



        register TOKEN  *old_head;

        while (head)
        {
                switch (head->tok)
                {
                case CCL:
                case NCCL:
                        free(head->string);
                        /* no break, fall through to default */

                default:
                        old_head = head;
                        head = head->next;
                        free(old_head);
                        break;
                }
        }
}
```

End Listing Two

(Listing three begins on page 75)

Changing Your Address?

Staple your label here.

To change your address, attach your address label from the cover of the magazine to this coupon and indicate your new address below.

Name

Address

Address Apt. #

City State

Zip

Mail to: DDJ, 2464 Embarcadero Way, Palo Alto, CA 94303

GREP.C (Listing Continued, text begins on page 50)
Listing Three

```
/*--------------------------------------------------------------------
 *         GREP.C: A generalized regular expression parser.
 *
 *                 Copyright (c) 1984 Allen Holub
 *      Copyright (c) 1984 Software Engineering Consultants
 *                        P.O. Box 5679
 *                     Berkeley, CA,  94705
 *
 *                     All rights reserved.
 *
 *      This program may be copied for personal,  non-commercial use
 *      only, provided that this copyright notice is included in all
 *      copies  and  that  this  program is not modified in any way.
 *      Copying for any other use without previously  obtaining  the
 *      written permission of the author is prohibited.
 *
 *      Machine readable versions of  this  program may be purchased
 *      for $35 from  Software  Engineering  Consultants.  Supported
 *      disk formats are CP/M 8" SS/SD and PCDOS (v2.x) 5-1/4" DS/DD.
 *
 *--------------------------------------------------------------------
 */


#include "a:stdio.h"
#include "b:tools.h"

/*
 *      GREP
 *
 *      Search a file for a pattern.
 *
 *      The algorithm used here is essentially the algorithm in
 *      Software Tools in Pascal (pp 145f.). Though the routines have
 *      been changed somewhat to put them into good 'C'. See tools.c
 *      for details.
 *
 *      This program is a healthy subset of the UNIX program of the same
 *      name. The differences are as follows:
 *
 *              - the -s, -x and -b options are not supported.
 *              - the meta-characters ()+? are not supported.
 *
 *      usage is:
 *              grep [-vclnhyef] [expression] files ...
 *
 *      To compile with the Computer Inovations C86 compiler use:
 *
 *              cc1 grep
 *              cc2 grep
 *              cc3 grep
 *              cc4 grep
 *              cc1 tools
 *              cc2 tools
 *              cc3 tools
 *              cc4 tools
 *              link grep tools,,,c86s2s.lib
 *
 *      To compile with Aztec CII use:
 *
 *              cc1 -x4000 grep.c
 *              as  grep.asm
 *              cc1 -x4000 tools.c
 *              as  tools.asm
 *              ln  grep.o tools.o a:libc.lib
 */
```

(Continued on next page)

Listing Three

```
#define CPM      1               /*      Comment this out if you're compiling
                                  *      in an MSDOS system.
                                  */

#define MAXLINE  128             /*      Maximum size of an input line
                                  */

#define MAX_EXPR 64              /*      The maximum number of regular
                                  *      expressions separated by
                                  *      newlines or | allowed.
                                  */


/*      The following global flags are true if a switch was set
 *      in the command line, false otherwise.
 */

int     vflag, yflag, cflag, lflag, nflag, hflag, fflag;

main(argc,argv)
int     argc;
char    **argv;
{
        int     i, j, linenum, count;

        int     line[MAXLINE];
        int     numfiles;
        FILE    *stream;
        int     exprc;


        TOKEN   *exprv[MAX_EXPR];

        i = 1;

        if (argc < 2)
                abort( pr_usage(1) );

        if ( *argv[i] == '-')
        {
                /*
                 *      Expand the switches on the command line
                 */

                expand_sw( argv[i++] );

                if ( i == argc )
                        abort( pr_usage(1) );
        }

        /*      Get the pattern string.
         */

        if ( (exprc = get_expr( exprv, MAX_EXPR, &argv[i++])) == 0 )
                abort( pr_usage(2) );

        numfiles = argc - i;            /*  Get number of files left to
                                         *  process on the command line
                                         */

        fprintf(stderr,"(c) Copyright 1984, Software Engineering Consultants\n"
        do
        {
                if ( numfiles)
                {
                        stream = fopen( argv[i], "r");
```

```
                if (stream == NULL)
                {
                        fprintf(stderr, "Can't open %s\n", argv[i]);
                        continue;
                }
        }
        else
        {
                stream = stdin;
        }

        count = 0;
        linenum = 1;

        while ( fgets(line, MAXLINE, stream) )
        {
#ifdef CPM
                if (!fflag  ||  yflag )
                        stoupper(line);
#else
                if ( yflag )
                        stoupper(line);
#endif

                for( j = exprc ; --j >= 0 ; )
                {
                        if ( matchs(line , exprv[j]) )
                        {
                                count++;
                                pr_match(linenum, line, argv[i], 1,
                                                        numfiles);
                        }
                        else
                        {
```

(Continued on next page)

Listing Three

```
                                        pr_match(linenum, line, argv[i], 0,
                                                             numfiles);
                                }

                                linenum++;
                                cntrl_c();
                        }
                        if( lflag && count )
                                break;
                }
                pr_count( numfiles, argv[i], count );
                fclose (stream);

        } while (++i < argc);

        abort();
}

/*------------------------------------------------------------------------*/


pr_count( fcount, fname, count)
int     fcount, count;
char    *fname;
{
        /*      Process the -c flag by printing out a count and,
         *      if more than one file was listed on the command line,
         *      the file name too.
         */

        if (!cflag)
                return;

        if (fcount > 1)
                printf("%-12s: ", fname );

        printf( "%d\n", count );
}

/*------------------------------------------------------------------------*/


pr_match(linenum, line, fname, match, numfiles)
int     linenum, match;
char    *line, *fname;
{
        /*      If a match is found print the correct thing
         *      as specified by the command line switches.
         */

        char    buf[80];

        if (cflag)
                return;

        if ( (vflag && !match) ||  (!vflag && match) )
        {
                if (!hflag && ( (numfiles > 1) || lflag) )
                        printf("%s%s", fname, lflag ? "\n" : ":" );

                if (nflag)
                        printf("%03d:", linenum );

                if (!lflag)
                        printf("%s", line );
```

```
		}
}

/*-----------------------------------------------------------------*/

pr_usage(num)
int	num;
{


#ifdef DEBUG
	fprintf(stderr,"%d ", num);
#endif
	fprintf(stderr,"usage: grep [-cefhlnvy] [expression] <files ...>\n");
}

/*-----------------------------------------------------------------*/


abort()
{
	exit();
}

/*-----------------------------------------------------------------*/


expand_sw( str )
char	*str;
{
	/*	Set global flags corresponding to specific switches
	 *	if those switches are set
	 */

	vflag = 0;
	cflag = 0;
	lflag = 0;
	nflag = 0;
	hflag = 0;
	fflag = 0;
	yflag = 0;

	while (*str)
	{
		switch ( toupper(*str))
		{
		case '-':
		case 'E':
			break;

		case 'C':
			cflag = 1;
			break;

		case 'F':
			fflag = 1;

			break;

		case 'H':
			hflag = 1;
			break;

		case 'L':
			lflag = 1;
			break;

		case 'N':
			nflag = 1;
			break;
```

(Continued on next page)

Listing Three

```
                case 'V':
                        vflag = 1;
                        break;

                case 'Y':
                        yflag = 1;
                        break;

                default:
                        pr_usage(3);
                        abort();
                        break;
                }

                str++;
        }
}

/*------------------------------------------------------------------------*/

int do_or( lp, expr, max )
char    *lp;
TOKEN   **expr;
int     max;
{
        int     found;
        TOKEN   *pat;
        char    *op;

        found = 0;

        /*
         *      Extract regular expressions separated by OR_SYMs from

         *      lp and put them into expr. Extract only up to
         *      max expressions. If yflag is true map string to upper
         *      case first.
         */

        if( yflag )
                stoupper( lp );

        while ( op = in_string(OR_SYM, lp) )
        {
                if(found <= max && (pat = makepat(lp, OR_SYM)) )
                {
                        *expr++ = pat;
                        found++;
                }
                lp = ++op;

                if ( pat == 0 )
                        goto fatal_err;
        }

        if (found <= max  &&  (pat = makepat( lp, OR_SYM))  )
        {
                found++;
                *expr = pat;
        }


        if ( pat == 0 )
        {
```

(Continued on page 82)

Oct. 1984, No. 96

Dr. Dobb's Journal

Reader Service Card

A. Your primary job function: (Check one only)

- ☐ Company management (Pres., V.P., Treas., Owner, Gen. Mgr., Mktg. Dir.)
- ☐ Computer systems management (V.P. EDP., MIS Director, Data Processing Mgr., Data Communications Mgr., Network Planner)
- ☐ Engineering management (V.P. Engrg., Chief Engr., Tech. Director, Dir. R&D)
- ☐ Systems integrators (Systems Designer, Project Engr., Systems Application Engr., Technical Staff Members)
- ☐ Consultants (Computer/EDP/Data Communications Consultant)
- ☐ Educators (Educational users and instructors of computer technology)
- ☐ Systems/Programming specialists—mini-micro systems
- ☐ Other (Please specify)____________

B. Which languages are you MOST interested in?

☐ BASIC	☐ C	☐ PL/1
☐ Fortran	☐ LISP	☐ APL
☐ COBOL	☐ Prolog	☐ Logo
☐ Pascal	☐ Ada	☐ Smalltalk
☐ Modula-2	☐ Forth	Other ______

C. What is the operating system?

- ☐ CP/M (or derived)
- ☐ UNIX (or derived)
- ☐ MS-DOS (or derived)
- ☐ Other

D. Please indicate which of the following microcomputers you currently own and/or plan to buy in the next 12 months.

	Own	Plan to Buy
Apple	☐	☐
Commodore	☐	☐
Digital Equipment/DEC	☐	☐
Heath/Zenith	☐	☐
Hewlett-Packard	☐	☐
IBM	☐	☐
Macintosh	☐	☐
Radio Shack/Tandy TRS 80	☐	☐
Texas Instruments	☐	☐
Other (Specify)	☐	☐
None	☐	☐

E. What best describes the work you do with this microcomputer?

- ☐ Business functions
- ☐ Software/Hardware development
- ☐ Scientific/Engineering/R&D applications

F. Are you the decision maker or very influentia[l] in computer-related purchases at work?

☐ Yes ☐ No

G. Is your company a dealer, distributor, or systems house for microcomputers?

☐ Yes ☐ No

H. Subscriber

☐ Yes ☐ No

To obtain information about products or services mentioned in this issue, circle the appropriate number listed belo[w]. Use bottom row to vote for best article in issue. One card per person.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 24 25 26
28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53
55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80
82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107
109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134

Articles: 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213

Name ____________________ **Phone (____)** __________

Address ____________________

City/State/Zip ____________________

Oct. 1984, No. 96

Dr. Dobb's Journal

Reader Service Card

A. Your primary job function: (Check one only)

- ☐ Company management (Pres., V.P., Treas., Owner, Gen. Mgr., Mktg. Dir.)
- ☐ Computer systems management (V.P. EDP., MIS Director, Data Processing Mgr., Data Communications Mgr., Network Planner)
- ☐ Engineering management (V.P. Engrg., Chief Engr., Tech. Director, Dir. R&D)
- ☐ Systems integrators (Systems Designer, Project Engr., Systems Application Engr., Technical Staff Members)
- ☐ Consultants (Computer/EDP/Data Communications Consultant)
- ☐ Educators (Educational users and instructors of computer technology)
- ☐ Systems/Programming specialists—mini-micro systems
- ☐ Other (Please specify)____________

B. Which languages are you MOST interested in?

☐ BASIC	☐ C	☐ PL/1
☐ Fortran	☐ LISP	☐ APL
☐ COBOL	☐ Prolog	☐ Logo
☐ Pascal	☐ Ada	☐ Smalltalk
☐ Modula-2	☐ Forth	Other ______

C. What is the operating system?

- ☐ CP/M (or derived)
- ☐ UNIX (or derived)
- ☐ MS-DOS (or derived)
- ☐ Other

D. Please indicate which of the following microcomputers you currently own and/or plan to buy in the next 12 months.

	Own	Plan to Buy
Apple	☐	☐
Commodore	☐	☐
Digital Equipment/DEC	☐	☐
Heath/Zenith	☐	☐
Hewlett-Packard	☐	☐
IBM	☐	☐
Macintosh	☐	☐
Radio Shack/Tandy TRS 80	☐	☐
Texas Instruments	☐	☐
Other (Specify)	☐	☐
None	☐	☐

E. What best describes the work you do with this microcomputer?

- ☐ Business functions
- ☐ Software/Hardware development
- ☐ Scientific/Engineering/R&D applications

F. Are you the decision maker or very influenti[al] in computer-related purchases at work?

☐ Yes ☐ No

G. Is your company a dealer, distributor, or systems house for microcomputers?

☐ Yes ☐ No

H. Subscriber

☐ Yes ☐ No

To obtain information about products or services mentioned in this issue, circle the appropriate number listed belo[w]. Use bottom row to vote for best article in issue. One card per person.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 24 25 26
28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53
55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80
82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107
109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134

Articles: 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213

Name ____________________ **Phone (____)** __________

Address ____________________

City/State/Zip ____________________

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT #27346, PHILADELPHIA, PA.

POSTAGE WILL BE PAID BY ADDRESSEE

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS
Dr. Dobb's Journal

P.O. BOX 13851
PHILADELPHIA, PA 19101

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT #27346, PHILADELPHIA, PA.

POSTAGE WILL BE PAID BY ADDRESSEE

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS
Dr. Dobb's Journal

P.O. BOX 13851
PHILADELPHIA, PA 19101

Put Dr. Dobb's Under the Christmas Tree!

Treat your friend (or yourself!) to Dr. Dobb's Journal

Bound Volumes

Every Issue Available For Your Personal Reference.

Vol. 1 1976

The material brought together in this volume chronicles the development in 1976 of Tiny BASIC as an alternative to the "finger blistering," front-panel, machine-language programming which was then the only way to do things. This is always pertinent for bit crunching and byte saving, language design theory, home-brew computer construction and the technical history of personal computing.
Topics include: Tiny BASIC, the (very) first word on CP/M, Speech Synthesis, Floating Point Routines, Timer Routines, Building an IMSAI, and more.

Vol. 2 1977

1977 found **DDJ** still on the forefront. These issues offer refinements of Tiny BASIC, plus then state-of-the-art utilities, the advent of PILOT for microcomputers and a great deal of material centering around the Intel 8080, including a complete operating system. Products just becoming available for reviews were the H-8, KIM-1, MITS BASIC, Poly Basic, and NIBL. Articles are about Lawrence Livermore Lab's BASIC, Alpha-Micro, String Handling, Cyphers, High Speed Interaction, I/O, Tiny Pilot & Turtle Graphics, many utilities, and even more.

Vol. 3 1978

The microcomputer industry entered its adolescence in 1978. This volume brings together the issues which began dealing with the 6502, with mass-market machines and languages to match. The authors began speaking more in terms of technique, rather than of specific implementations; because of this, they were able to continue laying the groundwork industry would follow. These articles relate very closely to what is generally available today.
Languages covered in depth were SAM76, Pilot, Pascal, and Lisp, in addition to RAM Testers, S-100 Bus Standard Proposal, Disassemblers, Editors, and much, much more.

Vol. 4 1979

This volume heralds a wider interest in telecommunications, in algorithms, and in faster, more powerful utilities and languages. Innovation is still present in every page, and more attention is paid to the best ways to use the processors which have proven longevity—primarily the 8080/Z80, 6502, and 6800. The subject matter is invaluable both as a learning tool and as a frequent source of reference.
Main subjects include: Programming Problems/Solutions, Pascal, Information Network Proposal, Floating Point Arithmetic, 8-bit to 16-bit Conversion, Pseudo-random Sequences, and Interfacing a Micro to a Mainframe—more than ever!

Vol. 5 1980

All the ground-breaking issues from 1980 in one volume! Systems software reached a new level with the advent of CP/M, chronicled herein by Gary Kildall and others (**DDJ's** all-CP/M issue sold out within weeks of publication). Software portability became a topic of greater import, and **DDJ** published Ron Cain's immediately famous Small-C compiler—reprinted here in full!
Contents include: the Evolution of CP/M, and CP/M-Flavored C Interpreter, Ron Cain's C Compiler for the 8080, Further with Tiny BASIC, a Syntax-Oriented Compiler Writing Language, CP/M-to-UCSD Pascal File Conversion, Run-time Library for the Small-C Compiler and, as always, even more!

Vol. 6 1981

Microcomputing was entering a technical maturity in 1981, while continuing to break new ground. This volume includes Dr. Dobb's first all-FORTH issue and the first Dr. Dobb's "Clinic" columns. There is continued coverage of CP/M and Small-C development, along with J.E. Hendrix's Small-VM and Santa Barbara Tiny BASIC for 6809—all here in one giant volume.
Articles include: Pidgin—A Systems Programming Language, The Conference Tree, Write Your Own Compiler with META-4, several exciting Z80 utilties, North Star tidbits, and more!

YES! Please enclose a gift card from me, and send the following volumes of **Dr. Dobb's Journal** to:

Name________ Address________

City________ State____ Zip____

Payment must accompany your order.

Please charge my: ☐ Visa ☐ MasterCard ☐ American Express

I enclose ☐ Check/money order

Card #________ Expiration Date________

Signature________

Name________ Address________

City________ State____ Zip____

Vol. 1 ____	x	$23.75	= ____
Vol. 2 ____	x	$23.75	= ____
Vol. 3 ____	x	$23.75	= ____
Vol. 4 ____	x	$23.75	= ____
Vol. 5 ____	x	$23.75	= ____
Vol. 6 ____	x	$27.75	= ____
All 6 ____	x	$125.00	= ____

(A savings of over 15%!)

Sub-total $____

Postage & Handling ____
Must be included with order.
Please add $1.25 per book in U.S.
($2.00 each outside U.S.)

TOTAL $____

Mail to: Dr. Dobb's Journal, 2464 Embarcadero Way, Palo Alto, CA 94303
Allow 6–9 weeks for delivery.

Code: 3011

Circle **no. 79** on reader service card.

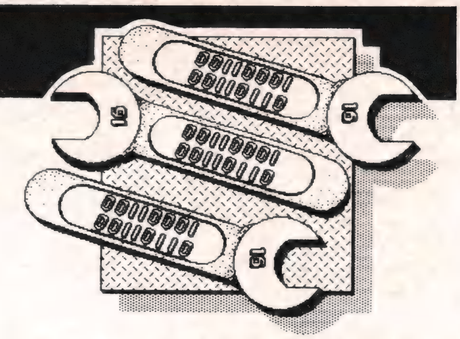
Listing Three

```
fatal_err:
                printf("Illegal expression\n");
                exit();
        }

        return (found);
}

/*------------------------------------------------------------------------*/

get_expr( expr, max,  defexpr )
TOKEN   *expr[];
int     max;
char    **defexpr;
{
        FILE    *stream;
        int     count;

        char    line[MAXLINE];

#ifdef DEBUG
        int     i;
#endif

        /*      Get regular expressions separated by | or newlines
         *      either out of a file or off the command line depending
         *      on whether the -f flag is set. The expressions are
         *      converted into pattern templates (see tools.c) and
         *      pointers to the templates are put into the array expr[]
         *      (which works similar to argv).
         *
         *      Return the number of expressions found (which can be used
         *      in a similar fashion to argc).
         */

        count = 0;

        if ( fflag )
        {
                /*
                 *      Then *defexpr is the file name and expressions should
                 *      be taken from that file.
                 */

                if ( (stream = fopen(*defexpr, "r")) == NULL )
                {
                        fprintf(stderr, "Can't open %s\n", *defexpr);
                        abort();
                }

                while ( (max - count)  &&  fgets(line, MAXLINE, stream) )
                {
                        count += do_or(line, &expr[count], max - count );
                }

                fclose (stream);
        }
        else
        {
                /*
                 *      *defexpr is the expression itself.
                 */

                if ( count += do_or( *defexpr, &expr[count], max - count))
                        *defexpr = " ";
        }
```

```
#ifdef DEBUG

        /*      Print out all the regular expressions after they have been
         *      converted into pattern templates (see tools.c).
         */

        for (i = count; --i >= 0 ; )
        {
                pr_tok(expr[i]);
                printf("--------------------------------------------------\n");
        }

#endif
        return(count);
}

/*--------------------------------------------------------------------------*/

cntrl_c()
{

#ifdef CPM

        /*      If any character was hit, and that character is a
         *      ^C, then abort.
         */

        if ( bdos(11) && ((bdos(1,0) & 0x7f) == 0x03) )
                abort();
#endif

}
```

End Listings

16-BIT SOFTWARE TOOLBOX



by Ray Duncan

MSDOS 2.0 Filters

One of the enhancements in MSDOS 2.0 is support for the concept of redirectable I/O á la Unix. Two predefined I/O channels, called the "standard input" and "standard output," may be accessed by any program; ordinarily they are directed to the keyboard and video display, respectively, but they can be individually redirected to other devices or to files with parameters placed in a DOS command line.

The standard I/O devices allow introduction of "filters," another feature of Unix, into the MSDOS environment. To quote the MSDOS manual, "A filter is a program or command that reads data from a standard input device, modifies the data, and writes the result to a standard output device . . . . By using the piping feature, you can cause a filter to receive its input from another command, or send its output to another command."

Three simple filters are supplied with MSDOS 2.0: SORT, which sorts text data; FIND, which searches an input stream to match a specified string; and MORE, which displays data one screen at a time. The witty technical writers at Microsoft go on to say, "You can easily add your own filter to the filters that have been supplied; just write a program that reads its input from the standard input device, and writes its output to the standard output device." You can easily build your own jetliner, too. Just hook some wings and a fuselage onto some engines, and there you are!

DDJ to the rescue once again. This month we present a detailed example of how to write a filter, with a little program called CLEAN. I previously published this program in a different form in *Softalk/ IBM*, and it was modified into a filter by Bob Taylor of Buffalo, New York. CLEAN can transform almost any kind of text input stream into a stripped-down output file. It may be used, for example, to massage a WordStar docume7< file or other word processing file into a form that EDLIN can cope with. It does this by stripping the high bit off of all characters, expanding tabs to spaces, and deleting all control codes except for form feeds, line feeds, and carriage returns.

In the command line for CLEAN, you must specify the source and destination for the text; otherwise it will simply read the default standard input (the keyboard) and write to the default standard output (the video display). For example, to filter the WordStar document file "MYFILE.WS" and leave the result in "MYFILE.TXT," you would enter:

```
A>CLEAN <MYFILE.WS
    >MYFILE.TXT
```

Note that the original file, MYFILE .WS, is unchanged. An invaluable application of the CLEAN filter is to rescue assembly language source files. I've found that when you accidentally perform extensive editing on such a source file in WordStar "document" mode instead of "nondocument" mode, the resulting file makes the assembler gag and spit out nasty messages. CLEAN lets you turn the source file back into something the assembler can swallow without losing all those painful hours of editing.

Another handy application for CLEAN is to list a WordStar document file in "raw" form on the printer, complete with print commands, etc.:

```
CLEAN <MYFILE.WS >PRN:
```

CLEAN is a simple program and highly modular, as you will see from the accompanying source code (Listing One, page 87). It illustrates reading from the standard input, writing to the standard output, and sending error messages to the (unredirectable) standard error device. It is also remarkably slow. This Macintosh-like performance can be ascribed to the fact that the input and output streams are being treated as character devices. Even when files are being processed, two calls are made to MSDOS for every character filtered. You will find that if you change the "get_char" and "put_char" routines to perform 1024-byte reads and writes, and block/deblock the data internally to the CLEAN program, performance becomes extremely spiffy.

Next month, we'll publish the much more sophisticated "TK" filter in this column. TK, contributed by Jim Mott, is a powerful token parser with many runtime options.

Sizing RAM under MSDOS

In a previous column (*DDJ* No. 90, April 1984), I published a rather circuitous method of finding the amount of available RAM under MSDOS 2.0. Several readers wrote in to take me to task for this item and showed me that there is a much more direct way to get the same result. Billy Smith's letter says it all:

"On the subject of sizing RAM under MSDOS, I know of a very simple solution that I think qualifies as machine independent. In appendix E of the DOS manual we have an explanation of how DOS prepares programs for execution. The preparation, besides loading the file itself into RAM, consists of:

(1) DOS setting up a PSP (Program Segment Prefix) just below the loaded program, and
(2) DOS initializing some of the registers.

"A careful examination of the structure of the PSP shows that the word at

PSP offset 2 contains the paragraph address of the first paragraph following the end of contiguous RAM. This is incredibly unclear in the manual, and it was only by test that I became certain of this fact.

"The manual fails to explicitly give the address of this value in the PSP. It must be deduced from an ambiguous diagram (page E-8). The value, which is called Top of Memory (not too bad a name), is footnoted as follows:

> First segment of available memory is in segment (paragraph) form (for example, hex 1000 would represent 64K).

"It seems that what they mean by 'available memory' is in fact 'unavailable memory,' i.e., the paragraph address of the first paragraph of memory past the end of contiguous RAM. At any rate, this fact of the DOS environment is readily available to every COM and EXE file at load time. I hope this makes sizing of memory a little easier for you and other readers of *DDJ* !"

Many thanks to Billy and the several other readers who provided this helpful information.

More Microsoft Assembler Warnings

While writing this month's column, I stumbled on a new discrepancy between the manual and the Macro Assembler. The manual leads you to believe that the following code is correct:

```
LES AX,DOUBLEWORD PTR
    [BP + 12]
```

In practice, that code causes the assembler to cough up a funny error message. What you need to write is:

```
LES AX,DWORD PTR [BP + 12]
```

The reserved token DWORD isn't mentioned anywhere in the manual as far as I can tell.

Bill Payne of Sandia Labs was kind enough to send a list of further "features" of the Microsoft Macro Assembler. These items were apparently collected on a CompuServe bulletin board by John Chapman and have passed through the hands of an unknown number of intermediate benefactors. I have verified them all again before this appearance in *DDJ*.

(1) The 8088 does not support a comparison of immediate data with a segment register. If you write
CMP ES,0
the assembler will produce the opcode for
CMP AX,0
and does not generate any error message.

(2) Most instructions with missing operands generate error messages; however, the instruction
MOV AX,
produces the machine code for
MOV AX,0
and no error message. Apparently, the same thing will happen for all instructions that can use a general register as the destination and immediate data as the source. Zero is always used for the missing operand.

(3) Erroneous code will be generated if square brackets denoting indexed addressing mode are omitted in certain operations, even though an error message would be expected. If you write
MOV BYTE PTR ES:DI,'$'

the assembler will generate the machine code for

MOV BYTE PTR ES: [7],'$'

rather than what was intended, which was

MOV BYTE PTR ES: [DI], '$'

This appears to occur because the assembler finds DI in its symbol table, equated to the register triplet 111B, and substitutes the 7 as though the programmer had written

DI EQU 7

(4) Even if a RADIX pseudo-op has been used to specify the default base for data values, the assembler will still check the last character for a valid radix specification and use it if present. For example,

```
.RADIX 16
SUB BX,0B
```

generates the machine code

83 EB 00

which is incorrect for X'0B', while

SUB BX,11d or SUB BX,0Bh

are both correct and will generate the machine code

83 EB 0B

which is the intended instruction.

(5) Data entry errors. Always scan comments in a newly entered program to make sure that each is preceded by a semicolon rather than a colon. The assembler will assume that the colon denotes a label and will generate the spurious error message "Open Procedures." Review labels in failing programs: the branch table sequence below provides both correct and incorrect examples.

```
          Correct code . : .
          JMP CS:JUMLIST[BX]
          . . .
          . . .
JUMLIST   DW routine1
          DW routine2
          Incorrect code . : .
          JMP CS:JUMLIST [BX]
JUMLIST:  DW routine1
          DW routine2
```

(6) Pseudo-operations. The XLIST pseudo-op will be ineffective during both passes if the command line parameter /D is specified for the assembly. Also, the assembler will not correctly resolve "identity" type definition errors in the EQUATE pseudo-op.

LOOPIT EQU LOOPIT
will cause the assembler to hang.

Hex to ASCII Conversion

Jim Garinger of Hermosa Beach, California, writes: "In the June *DDJ* 16-Bit Toolbox, you included a subroutine to convert a binary word to its hex ASCII equivalent. It seems to be typical of the methods used by my friends with 8-bit systems. Trying to do this same thing myself with the 8086 assembler instructions, I hit upon another way to do it and would like to share it with you. The listing is self-explanatory and lists the number of bytes of code and the clocks (you might want to check me on them) from the Intel manual, *IAPX 88 Book*. This, by the way, has proven to be my invaluable reference when doing 8086 assembler work.

"I would like to point out that further gains in speed can be made with this routine, using my original coding methods, by coding the call to hexbyte in line twice instead of as a near call. For the five bytes of added size penalty, there is a fair percentage of gain in clock speed. And nobody seems concerned about size anymore anyway.

"I did assemble both of our routines to check actual size, but I am afraid I can't guarantee the accuracy of the clocks or the addition (I ran out of fingers and toes early on)."

Jim's subroutine accompanies this column as Listing Two (page 90).

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 196.**

16-Bit Toolbox (Text begins on page 84)

Listing One

```
                                name    clean
                                page    55,132
                                title   'CLEAN - Filter text file'
                        ;
                        ; CLEAN - a DOS 2.0 filter for word processing document files.
                        ;
                        ; Originally written by Ray Duncan
                        ; Converted to DOS 2.0 filter by Bob Taylor.
                        ;
                        ; This program reads text from the standard input device and writes
                        ; filtered and transformed text to the standard output device.
                        ;
                        ;       1. High bit of all characters is stripped off.
                        ;       2. Tabs are expanded.
                        ;       3. Removes all control codes except for line
                        ;          feeds, carriage returns, and form feeds.
                        ;       4. Appends an end-of-file mark to the text, if
                        ;          none was present in the input stream.
                        ;
                        ; Can be used to make a WordStar file acceptable for
                        ; other screen or line editors, and vice versa.
                        ;
                        ;
 = 000D                 cr      equ     0dh             ; ASCII carriage return
 = 000A                 lf      equ     0ah             ; ASCII line feed
 = 000C                 ff      equ     0ch             ; ASCII form feed
 = 001A                 eof     equ     01ah            ; End-of-file marker
 = 0009                 tab     equ     09h             ; ASCII tab code

 = 0080                 command equ     80h             ; buffer for command tail

                        ; DOS 2.0 Pre-Defined Handles

 = 0000                 stdin   equ     0000            ; standard input file
```

(Continued on next page)

Listing One

```
        = 0001                  stdout  equ     0001            ; standard output file
        = 0002                  stderr  equ     0002            ; standard error file
        = 0003                  stdaux  equ     0003            ; standard auxilliary file
        = 0004                  stdprn  equ     0004            ; standard printer file

        0000                    cseg    segment para public 'CODE'

                                        assume  cs:cseg,ds:cseg

        0100                            org     100H            ; start .COM at 100H

        0100                    clean   proc    far             ; entry point from PC-DOS.
        0100  1E                        push    ds              ; push a long return back
        0101  33 C0                     xor     ax,ax           ; to DOS onto the stack.
        0103  50                        push    ax

        0104  E8 0160 R         clean3: call    get_char        ; get a character from input.
        0107  24 7F                     and     al,7fh          ; turn off the high bit.
        0109  3C 20                     cmp     al,20h          ; is it a control char?
        010B  73 1D                     jae     clean4          ; no. write it to output.
        010D  3C 1A                     cmp     al,eof          ; is it end of file?
        010F  74 4B                     je      clean6          ; yes, go write EOF mark and exit.
        0111  3C 09                     cmp     al,tab          ; is it a tab?
        0113  74 2D                     je      clean5          ; yes, go expand it to spaces.
        0115  3C 0D                     cmp     al,cr           ; is it a carriage return?
        0117  74 08                     je      clean35         ; yes, go process it.
        0119  3C 0A                     cmp     al,lf           ; is it a line feed?
        011B  74 04                     je      clean35         ; yes, go process it.
        011D  3C 0C                     cmp     al,ff           ; is it a form feed?
        011F  75 E3                     jne     clean3          ; no. discard it.
        0121                    clean35:
        0121  C7 06 0193 R 0000         mov     column,0        ; if it's a legit ctrl char,
        0127  EB 05 90                  jmp     clean45         ;  we should be back at column 0.

        012A  FF 06 0193 R      clean4: inc     column          ; if it's a non-ctrl char,
        012E                    clean45:                        ;  col = col + 1.
        012E  E8 0178 R                 call    put_char        ; write the char to output.
        0131  73 D1                     jnc     clean3          ; if OK, go back for another char.

        0133  BB 0002                   mov     bx,stderr       ; not OK.  Set up to show error.
        0136  BA 0195 R                 mov     dx,offset err_msg
        0139  B9 0018 90                mov     cx,err_msg_len  ; error = Disk full.
        013D  B4 40                     mov     ah,40h          ; write the error message
        013F  CD 21                     int     21h             ; to the standard error device. (CON:)
        0141  CB                        ret                     ; back to DOS.

        0142  A1 0193 R         clean5: mov     ax,column       ; tab code detected, must expand
        0145  99                        cwd                     ; expand tabs to spaces.
        0146  B9 0008                   mov     cx,8            ; divide the current column counter
        0149  F7 F9                     idiv    cx              ; by eight...
        014B  2B CA                     sub     cx,dx           ; eight minus the remainder is the
        014D  01 0E 0193 R              add     column,cx       ; number of spaces to send out to
        0151                    clean55:                        ; move to the next tab position.
        0151  51                        push    cx
        0152  B0 20                     mov     al,20h
```

```
0154 E8 0178 R                call   put_char        ; send an ASCII blank
0157 59                       pop    cx
0158 E2 F7                    loop   clean55
015A EB A8                    jmp    clean3

015C E8 0178 R        clean6: call   put_char        ; write out the EOF mark,
015F CB                       ret                    ; and return to DOS.

0160                  clean   endp


0160                  get_char proc near
0160 BB 0000                  mov    bx,stdin                  ; get chars from std. input
0163 B9 0001                  mov    cx,1                      ; # of chars to get = 1
0166 BA 0191 R                mov    dx,offset input_buffer    ; location = input_buffer
0169 B4 3F                    mov    ah,3fh
016B CD 21                    int    21h                       ; do the function call
016D 0B C0                    or     ax,ax                     ; test # of chars returned
016F 74 04                    jz     get_char1                 ; if none, return EOF
0171 A0 0191 R                mov    al,input_buffer           ; else, return the char in AL
0174 C3                       ret
0175                  get_char1:
0175 B0 1A                    mov    al,eof                    ; no chars read, return
0177 C3                       ret                              ; an End-of-File (EOF) mark.
0178                  get_char endp
```

(Continued on next page)

Listing One

```
0178                     put_char proc near
0178  A2 0192 R                   mov     output_buffer,al        ; put char to write in buffer.
017B  BB 0001                     mov     bx,stdout               ; write to std. output
017E  B9 0001                     mov     cx,1                    ; # of chars = 1
0181  BA 0192 R                   mov     dx,offset output_buffer ; location = output_buffer
0184  B4 40                       mov     ah,40h
0186  CD 21                       int     21h                     ; do the function call
0188  3D 0001                     cmp     ax,1                    ; check to see it was really done.
018B  75 02                       jne     put_char1
018D  F8                          clc                             ; really done. return carry = 0
018E  C3                          ret                             ; as success signal.
018F                     put_char1:
018F  F9                          stc                             ; not really done. return carry = 1
0190  C3                          ret                             ; as error signal (device is full).
0191                     put_char endp

0191  00                 input_buffer    db      0
0192  00                 output_buffer   db      0

0193  0000               column          dw      0

0195  0D 0A              err_msg         db      cr,lf
0197  63 6C 65 61 6E 3A                  db      'clean: Disk is full.'
      20 44 69 73 6B 20
      69 73 20 66 75 6C
      6C 2E
01AB  0D 0A                              db      cr,lf
= 0018                   err_msg_len     equ     (this byte)-(offset err_msg)

01AD                     cseg    ends

                                  end     clean
```

End Listing One

16-Bit Toolbox

Listing Two

Jim Garinger's binary to hex ASCII subroutine.

```
; Routine to convert 16-bit binary
; word to hexadecimal ASCII string.

; Call with:     AX = 16-bit binary value
;                SI = first-byte-address for
;                     resulting ASCII string
;
; Returns:       nothing
```

```
;
; Destroys:     AX, BX, CX, SI, clears direction flag
;
; In each comment, the first number is the number of
; bytes of machine code, and the second is the number
; of machine cycles to execute that instruction.

hexconv  proc     near
                             ;45 bytes/276 clocks
         cld                 ; 1   2
         mov      cl,4       ; 2   4 shift count
         mov      bx,offset hextbl
                             ; 3  12 point to table for translations
         mov      ch,al      ; 2   2 store for converting 2nd half

                             ; convert upper half of
                             ; word to hex ASCII

         call     hexbyte    ; 3  19  (98 clocks)
         mov      ah,ch      ; 2   2 get the word

                             ; convert lower half of
                             ; word to hex ASCII

         call     hexbyte    ; 3  19  (98 clocks)
         ret                 ; 1  2Ø
                             ;    8Ø (+ 2*98 = 276 total clocks)
hexbyte  proc     near
         xor      al,al      ; 2   3 clear for positioning
         shr      ax,cl      ; 2  2Ø low nibble into al
         shr      al,cl      ; 2  2Ø position lower nibble
         xlat     hextbl     ; 1  11 convert lower nibble
         xchg     al,ah      ; 2   2 get higher nibble
         xlat     hextbl     ; 1  11 convert higher nibble
         stosw               ; 1  11
         ret                 ; 1  2Ø

                             ;conversion table to hex ASCII
hextbl   db       "Ø123456789ABCDEF"

hexbyte  endp
hexconv  endp
```

End Listings

C/UNIX PROGRAMMER'S NOTEBOOK

By Anthony Skjellum

The thrust of this column is to provide examples of scientific uses for the C programming language. My aim is to present routines useful in conjunction with scientific programming; some are general and others implement specific numerical algorithms. I intend this column for several specific audiences. The first audience comprises those die-hard programmers who continue to produce useful programs in outdated languages. Not only is this of greater expense to themselves, but it also cheats others by locking them into Fortran or other old-fashioned languages. For them, I want to illustrate the elegance and versatility of C.

The second audience (which may also include the first) is those users interested in scientific applications who may not have used C for this purpose. This column should demonstrate that C is completely acceptable for such purposes, and the code shows how generally concepts can be presented. (I make no attempt to survey scientific applications where C could be used but merely include some nontrivial examples.) Finally, for those readers who don't fit into the above categories, the general purpose routines should still prove interesting.

The column presents three programming systems. The first is a set of general purpose subroutines designed to simplify the process of user – program interaction and to provide a straightforward means for handling erroneous input. The input mechanisms are not particularly sophisticated but emphasize structure in the user's program.

The routines make range checking so automatic that the programmer has no excuse for omitting such checks, regardless of how quickly a program must be completed. Providing these routines to novice programmers in a classroom environment has eliminated a lot of frustration over using the scanf( ) function.

The second and third programming systems illustrate Runge-Kutta integration. The Runge-Kutta formalism is a standard numerical technique for handling the numerical integration of one or more first-order ordinary differential equations. Interested readers may wish to consult the book *Numerical Analysis* by Richard L. Burden, et al. (Prindle, Weber, Schmidt Publishers, 2nd ed., 1981). This is the source for the algorithms presented in the code and is also a fine reference for introductory numerical methods. I chose the Runge-Kutta routines as examples because of their widespread use in scientific work.

Acknowledgements

The general purpose library has received extensive use by Caltech students during the past school year. Thanks are due to Scott Lewicki who discovered a couple of minor details that caused major errors.

The Runge-Kutta code was developed by Michael J. Roberts and my-

```
int iinp(prompt,cflag,low,high);
char *prompt:  optional prompt string to be printed before input
char cflag:    checking flag: if non-zero, range checking is performed
int low
int high:      low, high are (inclusive) range checking values
iinp( ) repeats input until a valid number is entered; the valid number is the func-
tion's return value.

double finp(prompt, cflag, low, high);
char *prompt:  optional prompt string to be printed before input
char cflag:    checking flag: if non-zero, range checking is performed
double low
double high:   low, high are (inclusive) range checking values
finp( ) repeats input until a valid number is entered; the valid number is the func-
tion's return value

len = sinp(prompt, string, length);
int len:       length of entered string
char *prompt:  optional prompt string to be printed before input
int length:    maximum length of input string
sinp( ) ignores leading spaces.

retn = cinq(prompt);
int retn:      1 —> 'Y' was typed, 0 —> 'N' was typed
char *prompt:  optional prompt string to be printed before input

retn = display(fname);
int retn:      0 —> success, -1 —> failure
char *fname:   null-terminated name of file
display( ) prints the specified file on the standard output device.
```

Table I
GPR Calling Sequences

self. Mr.Roberts developed the major portion of the RKSYS (multiple equations) routines as a project for Caltech's Introduction to Computational Physics course. Approximately 60 hours were spent developing, testing, and debugging this code. Mr. Roberts also wrote the original version of the documentation for RKSYS, included in modified form as Table III (page 94).

GPR: General Purpose Routines

The general purpose library consists of five subroutines. Four of these subroutines deal with input. The fifth is a simple facility for printing files to the console. This latter routine, called display( ), will be considered separately from the input functions.

The other routines are iinp( ), finp( ), sinp( ), and cinq( ). The first three provide integer, floating-point, and string input, respectively. The last is a yes-no question processor. Table I (page 92) provides the exact calling sequences for each of the routines.

Traditionally user input consists of a sequence of lines, such as the sequence for inputting an integer shown in Figure 1 (at right).

Entering this sequence repeatedly can be rather tedious, so error checking is often omitted. This practice leads to programs that do not handle user mistakes intelligently. The GPR routines allow you to replace the above sequence with a single line of code:

```
input = iinp ("Enter input variable
    —>",1,MIN,MAX);
```

Since using the GPR input functions is easier than using scanf( ), this variety of function should be well received and may be used in lieu of scanf( ) for most purposes. A further advantage of these functions is that they unclutter the user's program. More sophisticated checking must still be included explicitly; the input functions only do range checking.

The display( ) function was included to encourage users to provide on-line help/documentation along with their programs. This function allows users to print out additional text whenever appropriate, creating a trivial on-line help facility; a help feature is almost always appropriate but usually is omitted.

```
#define MIN 10
#define MAX 100
. . .
int input;
. . .
while (1)          /* input loop */
{
        printf("Enter input variable —> ");
        if (scanf("%u",&input) != 1)      /* get variable */
        {
                drain( );      /* drain spurious characters */
                continue;     /* skip range checks */
        }
        /* do range checking */
        if ((input >= MIN) && (input <= MAX))
                break;         /* we are done */
        printf("\nNumber out of range\n");
} /* keep looping until scanf( ) can read a variable */
```

Figure 1

File: RK4.C Subroutine library: Listing Three.

The C language call is as follows:

```
rk4(function,a,b,n,alpha,t,w);
```

where:

function	returns the righthand side f(y,t) of the system
a	is the start of the interval of integration
b	is the end of the interval of integration
n	is the number of integration steps
alpha	is the initial value y(0) = alpha
t	is the array where the times will be stored
w	is where the approximations to y will be stored

The formal C definitions for the functions and its parameters are:

1. rk4( ): n step integrator

```
rk4(function,a,b,n,alpha,t,w)
double (*function)( );    /* function giving f(t,y) */
double a;                 /* beginning of interval */
double b;                 /* end of interval */
int n;                    /* number of steps in interval */
double alpha;             /* initial condition for y */
double t[ ];              /* array for returning T[i] values */
double w[ ];              /* array for returning W[i] values */
```

2. rk4_1( ): 1 step integrator

```
rk4_1(function,h,time,yapprox)
double (*function)( );    /* pointer to function to integrate */
double h;                 /* step size */
double time;              /* current time step */
double *yapprox;          /* current approximation of function */
```

Table II
RK4: Runge-Kutta Integrator

Listing One (page 96) shows the GPR routines. The current list is not exhaustive but intended only to suggest a trend for additional routines.

RK4: Runge-Kutta Algorithm

Now we will consider a scientific application of C: single equation Runge-Kutta integration. Before introduction of the code, some background is required.

We start with a differential equation in the canonical form

$$y' = f(y,t)$$

where y′ is the first derivative of y with respect to t and f(y,t) is a piece-wise continuous function of its arguments. In addition to the differential equation, we are given an initial condition

$$y(t=0) = y0$$

where y0 is some real constant (e.g., 35.1). These two equations uniquely specify the solution y(t), which is as yet unknown. The need for numerical techniques arises when the differential equation cannot be solved analytically.

Many possible numerical approaches can solve this equation, but considering all of them is beyond the scope of the current discussion. It is sufficient to state that a technique called RK4 (Runge-Kutta fourth order) will solve the equation numerically with known error characteristics. Listing Two (page 98) presents the solution of a typical equation

$$y'(t) = 1 + y$$

with the initial condition

$$y(t = 0) = 5.0$$

Since this equation can also be solved analytically, a comparison with the exact solution will demonstrate the error characteristics of the method. The analytical solution turns out to be

$$y(t) = t + 5.0*\exp(t)$$

This result can be deduced by inspection.

The Runge-Kutta algorithm and code are presented in Listing Three (page 101). Readers interested in more information should consult the Burden book. Calling sequences are described in Table II (page 93).

RKSYS: Systems of Differential Equations

Imagine now that we hâve a set (system) of N first-order ordinary differential equations:

$$y'_i(t) = f_i(t, y_1, y_2 \ldots y_N)$$
$$(i = 1,2, \ldots N)$$

This problem is useful for solving other systems too, since sets of linear differential equations involving higher order derivatives can be transformed to larger systems of the above variety (see Burden). Thus, a program that can solve the above system has reasonably wide applicability.

Unfortunately, the problem is more complicated for N equations than for one equation, as is evident from Table III (page 95). This table describes the more general Runge-Kutta software, and Listing Four (page 106) contains the actual code. Listings Five and Six (pages 110–114) are example programs that use rk4n( ) to solve small systems of equations. Listing Five implements the same problem as Listing Two but uses rk4n( ) instead of rk4( ) to perform the integration.

Using the Integrator

The rk4n( ) subroutine will solve a sys-

```
Files:      RKS.C       Subroutine library:  Listing Four
            RKST1.C     Test program 1:      Listing Five
            RKST2.C     Test program 2:      Listing Six
The format of the C language call is as follows:
      rk4n(function,wsource,wstore,m,a,b,n,alpha,t,kuttas);
where:
      function   is a pointer to the function that will return the first derivatives
                 of each function in your system. It will be called as:
                       function(j,i,tval,rk_comp)
                 where:
                       j          is the current time step
                       i          is the current function number (0, 1, . . . ,
                                  n - 1).
                       tval       is the current time value
                       rk_comp    is a pointer to a function that your derivative
                                  function must call as:
                                        rk_comp(n,j,i)
                                  where:
                                        n is the function number in the system
                                          (0, . . . n - 1) of the function you
                                          wish  to evaluate
                                        j is the time step
                                        i is the current function number
      wsource    is a pointer to the function that will return a W value (which will
                 have been stored by your WSTORE routine—one need worry
                 only about storing and returning these values, not the values
                 themselves). It will be called as:
                       wsource(j,i)
                 where:
                       j      is the current time step
                       i      is the current function number
      wstore     is a pointer to the function that will store values of W (see
                 wsource above). It is called as:
                       wstore(j,i,value)
                 where:
                       j      is the current time step
```

Table III

(Continued on next page)

tem of first-order differential equations as specified by the calling program, using the Runge-Kutta fourth-order integrator (described in Burden, pages 239 – 240; refer also to page 205).

The calling program must provide information for the rk4n( ) subroutine before it can solve the system of differential equations. This information consists of:

- The first derivatives of each of the functions in the system
- Subroutines to store and retrieve values as the functions are integrated
- Initial conditions for each of the functions
- The range over which the functions will be integrated
- Storage areas for the time and intermediate "k" value arrays

The information must be provided in a specific order and format, as described in Table III.

By studying the Runge-Kutta routines, the reader will notice the central importance of pointers to functions in the organization of the code. Using this concept effectively allows the software to be completely divorced from the specifics of the user's program.

Conclusions

This article presented three sets of example routines to demonstrate how scientific applications and related software are actually implemented in C. Studying the examples should help the reader to gain insight into the use of C for similar undertakings. DDJ

(Listings begin on page 96)

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 197**.

	i is the current function number
	value is the value to be stored for that location
	Note: wsource(j,i) should be equal to "value" after wstore(j,i,value).
m	is the number of equations in your system
ε	is the starting time value for the interval
b	is the ending time value for the interval
n	is the number of points into which the interval is to be broken
alpha	is a one-dimensional array of the initial values. The value alpha[0] is the first function at time = a, alpha[1] is the second, etc.
t	is a one-dimensional array where rk4n( ) will store the time values as it calculates them. It must be at least as large as n.
kuttas	is a two-dimensional array, the size of whose second element is 4 (i.e., Kuttas[ ][4]). It will be the storage area for "k" values as they are calculated.

The formal C definitions for the function and its parameters are:

```
double rk4n(function,wsource,wstore,m,a,b,n,alpha,t,kuttas)
double (*function)( );
double (*wsource)( );
double (*wstore)( );
int m;
double a;
double b;
int n;
double alpha[ ];
double t[ ];
double kuttas[ ][4];
```

Comments on array sizes, (*wsource)( ), (*wstore)( ):
The (*wstore)( ) function should trap out-of-bound storage requests that result as a natural part of the rk4n( ) algorithm. A more elegant solution is to make the array size used by ((one greater than the number of steps.

Table III
RK System Solver (RKSYS)

C/Unix (Text begins on page 92)

Listing One

```
/*
        gpr.c                                   created: 01-Oct-83

        general purpose utility routines for use with C to simplify
        user/program interaction.

        by Anthony Skjellum

        Copyright 1983, 1984 (c) Caltech.  All rights reserved.
        This subroutine library may be distributed freely and
        used for all non-commercial purposes but may not be sold.

        Routines:

                iinp():  integer input w/ prompt, range check + retry
                finp():  float   input w/ prompt, range check + retry
                sinp():  string  input w/ prompt
                cinq():  yes/no question processor w/ prompt + retry
                display(): display a file

        updated: 20-Jul-84

        testing information: this code was tested with Aztec C 1.05j
*/

#include <stdio.h>

/* Unix flag (used by display()) */
/*
#define UNIX 1
*/
#ifndef UNIX
#define TEOF    26      /* ^Z */
#endif

/* general purpose subroutines: */

/* iinp(): integer input with range checking, prompt and retry */

iinp(prompt,cflag,low,high)
char *prompt;
char cflag;
int low;
int high;
{
        int ival;

        while(1)
        {
                printf("%s",prompt);
                if(scanf("%d",&ival) < 1)
                        while(getchar() != '\n')
                                ;

                if((!cflag)||(ival >= low)&&(ival <= high))

                        break;
                                /* no checking, or within bounds */

                printf("\nValue out of range, try again...\n");
        }

        return(ival); /* return the value */

} /* end iinp() */
```

```
/* finp(): floating point input with range checking, prompt and retry */

double finp(prompt,cflag,low,high)
char *prompt;
char cflag;
double low;
double high;
{
        double fval;

        while(1)
        {
                printf("%s",prompt);
                while(scanf("%lf",&fval) < 1)
                        while(getchar() != '\n')
                                ;

                if((!cflag)||(fval >= low)&&(fval <= high))
                        break;
                                /* no checking, or within bounds */

                printf("\nValue out of range, try again...\n");
        }

        return(fval); /* return the value */

} /* end finp() */

/* subroutine sinp(): input a string with prompt, length limit */

sinp(prompt,string,length)
char *prompt;
char *string;
int length;
{
        int len;                        /* length of actual string input */
        char chr;
        char *fgets();                  /* string input function */

        printf("%s",prompt);                    /* display the prompt */
        while(isspace(chr = getchar()))
                ;
        ungetc(chr,stdin);
        fgets(string,length,stdin);     /* input the string */

        if((len = strlen(string)))
                string[strlen(string)-1] = '\0';

        return(len);
} /* end sinp() */

/* subroutine cinq(): yes no question processor with prompt, retry  */

cinq(prompt)
char *prompt;
{
        char chr;

        while(1)
        {
                printf("%s",prompt);

                do /* drain spurious 'white space' */
                {
                        chr = tolower(getchar()); /* use first char */
                }
                while(isspace(chr));

                if((chr == 'y')||(chr == 'n')) break;

                printf("\nRespond with Y or N, please try again...\n");
        }

        return((chr == 'y') ? 1 : 0);
}
```

(Continued on next page)

Listing One

```
/* display(): subroutine to print an ascii file on the console */

display(fname)
char *fname;
{
        char c;         /* character to output */
        FILE *disp;

        if((disp = fopen(fname,"r")) == NULL)
                return(-1);     /* can't open file */

        while((c = getc(         /* And the last one */

        *yapprox += (k1 + 2.0*(k2 + k3) + k4)/6.0; /* new approx */
}
```

End Listing One

Listing Two

```
/*
        program:        rktest1.c
        created:        03-Nov-83
        by:             A. Skjellum

        Copyright 1983, 1984 (c) California Institute of Technology.
        All rights Reserved.  This program may be freely distributed
        for all non-commercial purposes but may not be sold.

        updated:        16-Nov-83
        purpose:        illustrate the use of rk4 program

        uses:           rk4.c

        summary:
                integrates the differential equation:

                y'(t) + y(t) = t + 1
                y(0) = 5.0.

                for which the exact solution:

                y(t) = t + 5exp(-t) is known.
*/

/* constants */

#define YZERO   5.0     /* initial value for y */
#define TSTART  0.0     /* starting time for integration */
#define TEND    10.0    /* ending time for integration */
#define STEPS   80      /* 40 steps in integration */


/* subroutines: */

/* exact(): returns exact solution value, given t */

double exact(t)
double t;
{
        extern double exp();    /* exponential function */
```

(continued on page 100)

Listing Two

```
        if(t)
                return((t + YZERO*exp(-t)));

        return(YZERO);
}

/* fn(t,y): return f(t,y) given t,y values */

double fn(t,y)
double t;
double y;


{
        /*
                differential equation is y' + y = t + 1

                therefore, f = t + 1 - y.

        */

        return(t + 1.0 - y);
}

/* solutn(): print solution step at console */

solutn(t,y)
double *t;      /* pointer to t value */
double *y;      /* pointer to y value */
{
        printf("t = %7.3e, y = %7.3e, y_exact = %7.3e, diff = %7.3e\n",
                *t,*y,exact(*t),*y - exact(*y));
}

/* main program: */

main()
{
        /* external declarations */

        double fn();    /* ensure that this is typed as double */

        /* local variables: */

        register int i;

        double yarray[STEPS],tarray[STEPS];
                /* integrated solution stored here */


        /* begin code: */

        printf("\n\nrktest1.c   as of 03-Nov-83\n\n");
        printf("Integrates: y' + y = 1 + t     for\n\n");
        printf("t = %7.3e to %7.3e, with %u steps\n\n",
                TSTART,TEND,STEPS);

        /*
                integrate the answer from t = 0 to t = 10 sec
                80 points.
        */

        rk4(fn,TSTART,TEND,STEPS,YZERO,tarray,yarray);
                        /* compute the answers */
```

```
        for(i=0;i<STEPS;i++)     /* print solution */
                solutn(tarray+i,yarray+i);

        printf("\n\nEnd of execution\n\n");
}
```

End Listing Two

Listing Three

```
/*
        Runge - Kutta order 4 Algorithm

        Creation date:  31-Oct-83
        Author:         Mike Roberts

        Copyright 1983, 1984 (c) California Institute of Technology.
        All rights Reserved.  This program may be freely distributed
        for all non-commercial purposes but may not be sold.

        This algorithm is described in detail on page 205 of
        Burden, Richard L.:  Numerical Analysis.

        To approximate the solution of the initial value problem
                y'=f(t,y),  a<=t<=b, y(a)=alpha,
        at (N+1) equally spaced numbers in the interval [a,b]:
        INPUT endpoints a,b; integerg N; initial condition alpha.
        OUTPUT approximation w to y at the (N+1) values of t.

Step 1:
        Set     h=(b-a)/N;
                t=a;
                w=alpha;
        Output (t,w).
Step 2:
        For i=1,2,...,N do Steps 3-5:

        Step 3:
                Set     K1=hf(t,w);
                        K2=hf(t+h/2,w+K1/2);
                        K3=hf(t+h/2,w+k2/2);
                        K4=hf(t+h,w+K3).
        Step 4:
                Set w=w+(K1+2K2+2K3+K4)/6;      (Compute w[i].)
                        t=a+ih.                 (Compute t[i].)
        Step 5:
                Output (t,w).

Step 6:
        Stop.


*/


#define FALSE 0
#define TRUE 1

rk4(function,a,b,n,alpha,t,w)

double (*function)();   /* function giving f(t,y) */
double a;               /* beginning of interval */
double b;               /* end of interval */
int n;                  /* number of steps in interval */
double alpha;           /* initial condition for y */

double t[];             /* array for returning T[i] values */
double w[];             /* array for returning W[i] values */

 {
        register int i; /* counter for integration steps */
        double h;       /* stepsize */
        double time;
        double yapprox; /* approximation for y value  */
```

(Continued on next page)

Serious Software For The C Programmer From Lifeboat.™

Lattice® C Compiler: *The serious software developer's first choice.*

Selected for use by IBM,® Texas Instruments, Wang,® MicroPro,® Ashton-Tate,™ IUS/Sorcim,® Microsoft® and Lotus™ to name a few of the many. Why?

Lattice C is clearly the finest 16 bit C compiler available today.

—Renowned for speed and code quality.
—Fully compatible with the C standards set forth by Kernighan and Ritchie.
—Four memory model options offer you unsurpassed control and versatility.
—Superior quality documentation.
—Now includes automatic sensing and use of the 8087 chip.
—Widest selection of supporting add-on packages.

Halo™: *A graphics development package rapidly emerging as the industry standard.*

—140 graphics commands including plot, line, arc, box circle and ellipse primitives, bar and pie charts; pattern fill and dithering commands.
—New: multiple viewports and "stroke text" for angling, scaling and filling text.

C Food Smorgasbord™: *This beautifully written collection of C functions is a valuable time saver.*

—Library includes a binary coded decimal arithmetic package, level 0 I/O functions, a terminal independence package, IBM PC ROM BIOS access functions and much more.

Pmate™: *The premier editor for the programming professional.*

Pmate is a full screen editor with its own powerful macro command language:

—Perform on screen row and column arithmetic, alphabetize lists, translate code from one language to another, call up other macros.
—Customize Pmate almost any way you like.
—Contains 10 auxiliary buffers for storage of macros, text, subroutines.
—An "undo" feature allows the programmer to retrieve whole series of deleted items.

***Additional C Tools Available From** Lifeboat:*

Panel™: Screen formatter and data entry aid.
Lattice Windows™: Windowing utility; create "Virtual Screens."
Plink-86™: The popular linker; includes extensive overlay capabilities.
Pfix86™: Dynamic debugging utility.
Pfix86 Plus™: Symbolic debugger with capacity to debug overlays.
Btrieve™: Database record access/retrieval library.
Phact: Multikeyed ISAM C-Function library.
Fabs: Fast access B-tree database function library.
Autosort: Fast sort/merge utility.
ES/P: 'C' program entry with automatic syntax checking and formatting.
Greenleaf Functions™: Library of over 200 popular C functions.
And much more.

YES! Please rush me the latest FREE Lifeboat™ catalog of C products.

*Company Name*________ *Business Phone*________

*Name*________ *Title*________

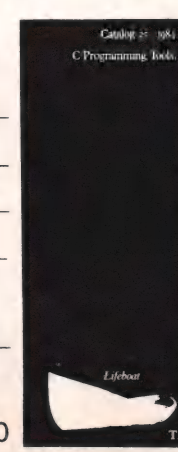
*Address*________

*City*________ *State*________ *Zip*________

Please check the category where Lifeboat can best help you:

☐ *Software development* ☐ *Corporate* ☐ *Education*
☐ *Dealer/distributor* ☐ *Government* ☐ *Other*________

Call Direct: 1-800-847-7078 (In NY State: 212-860-0300)
Return coupon to: Lifeboat Associates™
1651 Third Avenue, New York, NY 10128. DD/10



Circle **no. 42** on reader service card

Listing Three

```
        /*  STEP 1:  Initialization  */

        h = (b-a) / (double)n ;          /* Compute stepsize */
        time = a;                        /* Initialize time */
        yapprox = alpha;                 /* Start with the approximation
                                            equal to the initial value */

        for (i=0; i<n; i++)              /* Main integration loop */
        {
                if(i)   /* if not first time, call the integrator */
                        rk4_1(function,h,time,&yapprox);

                        /* Pass the function pointer, the h, time, and
                                yapprox values, and the pointers to
                                the current positions in the T and W
                                matrices */

                time = a + h*(double)i;  /* compute time */
                t[i] = time;             /* also save it */
                w[i] = yapprox;          /* store value for function */
        }
}

/*  This is the RK4 integrator portion.  It performs one step of the
        integration, and is called on each step from the RK4 loop.

        function = pointer to function to integrate
        h = stepsize
        time = current time location
        yapprox = current w (function approximation)

*/

rk4_1(function,h,time,yapprox)

double (*function)();            /* Pointer to function to integrate */
double h;                        /* Step size */
double time;                     /* Current time step */
double *yapprox;                 /* Current approximation of function */

{
        double k1, k2, k3, k4;  /* Temporary values in RK calculation */

        k1 = h * (*function)(time,*yapprox);    /* Evaluate first approx */
        k2 = h * (*function)(time+h/2.0, *yapprox+k1/2.0);
                                                /* Evaluate second approx *


        k3 = h * (*function)(time+h/2.0, *yapprox+k2/2.0);
                                                /* And the third */
        k4 = h * (*function)(time+h, *yapprox+k3);
                                                /* And the last one */

        *yapprox += (k1 + 2.0*(k2 + k3) + k4)/6.0; /* new approx */
}
```

End Listing Three

(Listing Four begins on page 106)

Listing Four

```
/*
        rk4n.c                          created: 07-Nov-83
        authors:                        A. Skjellum,
                                        M. Roberts

        updated:                        14-Nov-83
                                        by MJR

        Copyright 1983, 1984 (c) California Institute of Technology.
        All rights Reserved.  This program may be freely distributed
        for all non-commercial purposes but may not be sold.

        Purpose:
        integrate M first order differential equations

        y'[i] = f[i](t;y[j=1...M])

        M equations.

        algorithm:      see Burden, Faires, Reynolds, p. 239-240
                        also see p. 205

                1. interval t = [a,b]
                2. choose N > 0 as as partition of interval (N steps)
                3. define step size h = (b-a)/N.
                4. Initial conditions: (denote w[i,j] as approxes to y's)

                        w[i,0] = alpha[i]
                        means: ith w at time zero is set to initial value
                               alpha[i].

                5. Computing the w[i,j+1] from w[i,j] is done as follows:

                        loop over i = 1 to M

                            compute k1[i] = h*f[i](t,w[1,j],...,w[M,j])

                        end of loop

                        loop over i = 1 to M

                            compute k2[i] = h*f[i](t+h/2,w[1,j]+.5*k1[1],...,
                                            w[M,j] + .5*k1[M])

                        end of loop

                        loop over i = 1 to M

                            compute k3[i] =h*f[i](t+h/2,w[1,j]+.5*k2[1],...,
                                           w[M,j] + .5*k2[M])

                        end of loop

                        loop over i = 1 to M


                            compute k4[i] =h*f[i](t+h,w[1,j]+k3[1],...,
                                           w[M,j] + k3[M])

                        end of loop

                        loop over i = 1 to M

                            w[i,j+1] = w[i,j] +
                                       (k1[i] + 2*k2[i] + 2*k3[i] + k4[i])/6

                        end of loop
```

```
*/

double  (*rk_function)();
double  (*rk_source)();
double  (*rk_store)();
double  (*rk_kuttas)[4];
double  (*rk_comp[4])();                 /* tells us how to form k's */

/* functions called indirectly by k_calc()  */

double rk_1(n,j,i)       /* to provide compatibility with calling */
int n;
int j;
int i;
{
        return ((*rk_source)(j,n));
}

double rk_23(n,k,j,i)  /*  N is in { 0...M } = argument number.
                           Since we have one argument to FN per equation,
                           N will indicate which we are currently being
                           asked to provide.  Same goes for other RK_x
                           functions below.  */
int n;
int k;
int j;
int i;
{
        return((*rk_source)(j,n) + .5*rk_kuttas[n][k]);
}

double rk_2(n,j,i)
int n;
int j;
int i;
{
        return(rk_23(n,0,j,i));
}

double rk_3(n,j,i)
int n;
int j;
int i;

{
        return(rk_23(n,1,j,i));
}

double rk_4(n,j,i)
int n;
int j;
int i;
{
        return((*rk_source)(j,n) + rk_kuttas[n][2]);
}


/*  Here's the integrator!!! */


double rk4n(function,wsource,wstore,m,a,b,n,alpha,t,kuttas)

double (*function)();   /* pointer to function which returns deriv info */
double (*wsource)();    /* source of w[i,j] values */
double (*wstore)();     /* function which stores w[i,j] values for us */
int m;                  /* number of equations */
double a;               /* start of interval */
double b;               /* end   of interval */
int n;                  /* number of points */
double alpha[];         /* array of initial values */
double t[];             /* array where we store times */
double kuttas[][4];     /* n x 4 kuttas (k1,k2,k3,k4 i=1,...m) */
{
```

(Continued on next page)

Listing Four

```
        register int i; /* looping variable */
        register int j; /* looping variables */
        double time;
        double h = (b-a)/(double)n;     /* step size */

        double rk_k2(),rk_k3(),rk_k4(); /* include for emphasis */

        rk_function = function;

        rk_kuttas   = kuttas;
        rk_source   = wsource;
        rk_store    = wstore;

        rk_comp[0]  = rk_1;
        rk_comp[1]  = rk_2;
        rk_comp[2]  = rk_3;
        rk_comp[3]  = rk_4;

        /*  First assign initial values  */

        for (i = 0;i < m;i++)
                (*rk_store)(0,i,alpha[i]);

        /*  Now loop through the necessary loop, calculating
                each K value for each equation in each time step. */



        for(j = 0;j < n;j++)    /* n time steps */
        {
                time = a + h*(double)j;  /* compute time */
                t[j] = time;             /* also save it */

                rk4_1n(m,j,time,h);
        }
}

/* rk4_1n(): compute one solution step for n equations */

rk4_1n(m,j,time,h)
int m;
int j;                          /* time step we're working on */
double time;
double h;
{
        register int k; /* k calculation loop */
        register int i; /* m equations loop */
        double value;   /* temporary */

        for(k=0;k < 4;k++)      /* k compute loop */
                k_calc(m,k,j,time,h);

        for(i=0;i<m;i++)        /* compute new w[i,j]'s j fixed here */
        {
                value = (*rk_source)(j,i) + .166666667*(rk_kuttas[i][0] +
                        2.0*(rk_kuttas[i][1] + rk_kuttas[i][2])
                            + rk_kuttas[i][3]);
                                /* value for w[i,j] */
                (*rk_store)(j+1,i,value); /* save this hard got number */

        }
}

/* k_calc(): compute rk coefficients for fixed j */
```

(Continued on page 110)

Listing Four

```
k_calc(m,k,j,time,h)
int m;
int k;
int j;
double time;
double h;
{
        register int i;
        double tval;

        switch(k)
        {
                case 0:
                        tval = time;
                        break;
                case 1:

                case 2:
                        tval = time + .5*h;
                        break;
                case 3:
                        tval = time + h;
                        break;
        }

        for(i=0;i<m;i++)                    /*  used to be 1;<=m  */
        {
                rk_kuttas[i][k] = h*(*rk_function)(j,i,tval,rk_comp[k]);
        }
}
```

End Listing Four

Listing Five

```
/*
        program:        rkst1.c
        created:        03-Nov-83
        by:             A. Skjellum

        modified:       14-Nov-83
        by:             M. J. Roberts

        Copyright 1983, 1984 (c) California Institute of Technology.
        All rights Reserved.  This program may be freely distributed
        for all non-commercial purposes but may not be sold.

        purpose:        illustrate the use of rk4n program

        uses:           rk4n() (rks.c)

        summary:
                integrates the differential equation:

                y'(t) + y(t) = t + 1
                y(0) = 5.0.

                for which the exact solution:

                y(t) = t + 5exp(-t) is known.

                Integrates the same equation as rktest1
                but using the more general equation solver,
                rk4n().  This run is exactly the same as for
```

```
                rktest1, except that, rather than trying to solve
                exactly the same equation, we will solve a
                "system" of one differential equation.

*/

/* constants */

#define SYSIZE  1       /* number of functions in system */
#define YZERO   5.0     /* initial value for y */
#define TSTART  0.0     /* starting time for integration */
#define TEND    10.0    /* ending time for integration */
#define STEPS   50      /* 50 steps in integration */

/* variables external to all functions */

double wvalue[STEPS+1][SYSIZE];
double yarray[STEPS+1][SYSIZE];
double tarray[STEPS];
        /* integrated solution stored here */


/* subroutines: */

/* exact(): returns exact solution value, given t */


double exact(t)
double t;
{
        extern double exp();    /* exponential function */

        return((t + YZERO*exp(-t)));
}

/* fn(j,i,t,y): return f(t,y) given t,y values */

double fn(j,i,t,y)
int j;
int i;
double t;
double (*y)();
{
        double a,b;             /*  temporary storage space  */

        /*
                differential equation is y' + y = t + 1

                therefore, y' = t + 1 - y.

        */

        a = (*y)(0,j,i);        /*  calculate function
                                Note that the ZERO was passed so as to
                                allow the function to know which argument
                                we are talking about -- in this case,
                                we only need one argument evaluated, so
                                pass it 0 to indicate the first (zeroeth,
                                actually) argument is to be calculated.  */

        b = t + 1.0 - a;        /*  and figure out the rest of it  */
        return(b);

}


/* store():  the routine to store away the W values for later reference */

double store(row, col, value)
int row, col;           /*  location to store the value  */
double value;           /*  the actual value to store  */
{
```

(Continued on next page)

Listing Five

```
        wvalue[row][col]=value;
        return (value);
}

/* source():  return the  W value referenced by input parameters  */

double source(row,col)
int row, col;           /*  location to look up */


{
        return (wvalue[row][col]);
}

/* solutn(): print solution step at console */


solutn(j,i)
int j,i;        /* element numbers  */
{
        double time;
        double ex;
        double approx;

        time   = tarray[j];
        ex     = exact(time);
        approx = source(j,i);

        printf("t = %7.3e, y = %7.3e, y_exact = %7.3e, diff = %7.3e\n",
                time,approx,ex,approx - ex);
}

/* main program: */

main()
{
        /* external declarations */

        double store(), source();

        double fn();    /* ensure that this is typed as double */

        /* local variables: */

        register int i,j;


        double init[1];         /* initial condition matrix */

        /* begin code: */

        printf("\n\nrktest1.c   as of 03-Nov-83\n\n");
        printf("Integrates: y' + y = 1 + t     for\n\n");
        printf("t = %7.3e to %7.3e, with %u steps\n\n",
                TSTART,TEND,STEPS);

        /*
                integrate the answer from t = 0 to t = 10 sec
                STEPS points.
        */

        init[0] = YZERO;        /* set up initial condition matrix */

        rk4n(fn,source,store,SYSIZE,TSTART,TEND,STEPS,init,tarray,yarray);
                        /* compute the answers for 1 function */
```

(Continued on page 114)

Listing Five

```
        /*  Print out solution   */

        for(j=0;j<STEPS;j++)     /* print solution */
                solutn(j,0);
        ,

        printf("\n\nEnd of execution\n\n");
}
```

End Listing Five

Listing Six

```
/*
        program:        rkst2.c
        created:        03-Nov-83
        by:             A. Skjellum

        modified:       14-Nov-83
        by:             M. J. Roberts
        and:            5-Dec-83
        by:             M. J. Roberts

        modified:       25-Jul-84
        by:             A. Skjellum

        Copyright 1983, 1984 (c) California Institute of Technology.
        All rights Reserved.  This program may be freely distributed
        for all non-commercial purposes but may not be sold.

        purpose:        illustrate the use of RKS program
        update:         to test the rk4n() subroutine using a system
                        of two differential equations.

        uses:           rk4n() (rks.c)
        summary:

                integrates the differential equation system:

                u1'(t) = 8u2(t)                 u1(0) = 10
                u2'(t) = 2u1(t)                 u2(0) = 7

                for which the exact solution is known to be:

                u1(t) = 12exp(4t) - 2exp(-4t)
                u2(t) =  6exp(4t) +  exp(-4t)

*/

/* constants */

#define SYSIZE  2       /* number of functions in system */
#define Y1ZERO  10.0    /* initial value for first equation */
#define Y2ZERO  7.0     /* initial value for other equation  */
#define TSTART  0.0     /* starting time for integration */
#define TEND    1.0     /* ending time for integration */
#define STEPS   50      /* 50 steps in integration */

/* variables external to all functions */
```

```
double wvalue[STEPS+1][SYSIZE];
double yarray[STEPS+1][SYSIZE],tarray[STEPS];
        /* integrated solution stored here */


/* subroutines: */


/* exact(): returns exact solution value, given t */

double exact(n,t)
int n;          /*  which equation is it?  0 or 1?  */
double t;

{
        extern double exp();    /* exponential function */

        /*  This must find solutions for both U1 and U2.  The
                exact solutions are given in the header comments to
                this program, above.  */

        switch (n)
        {
                case 0:
                        return(12*exp(4*t) - 2*exp(-4*t));
                case 1:
                        return( 6*exp(4*t) +   exp(-4*t));
}       }


/* fn(j,i,t,y):  return f(t,y) given t,y values */

double fn(j,i,t,y)
int j;
int i;
double t;
double (*y)();
{
        switch (i)
        {
                case 0:
                        /*  u1'(t) = 8 * u2(t)   */
                        return(8*(*y)(1,j,i));

                case 1:
                        /*  u2'(t) = 2 * u1(t) */
                        return(2*(*y)(0,j,i));
        }
}


/* store():  the routine to store away the W values for later reference */

double store(row, col, value)
int row, col;           /*  location to store the value  */
double value;           /*  the actual value to store  */
{
        wvalue[row][col]=value;
        return (value);
}

/* source():  return the  W value referenced by input parameters  */


double source(row,col)
int row, col;           /*  location to look up */

{
        return (wvalue[row][col]);
}

/* solutn(): print solution step at console */
```

(Continued on next page)

Listing Six

```
solutn(j,i)
int j,i;        /* element numbers  */
{
        printf("\nt=%7.3e, y%1d=%7.3e, y%1d_exact=%7.3e, diff=%7.3e",
                tarray[j],i,source(j,i),i,exact(i,tarray[j]),
                source(j,i) - exact(i,tarray[j]));
}

/* main program: */

main()
{
        /* external declarations */

        double store(), source();

        double fn();    /* ensure that this is typed as double */

        /* local variables: */

        register int i,j;


        double init[SYSIZE];            /* initial condition matrix */

        /* begin code: */

        printf("\n\nrkst2.c     as of 25-Jul-84\n\n");
        printf("        Integrates the differential equation system:\n\n");
        printf("        u1'(t) = 8u2(t)                 u1(0) = 10\n");
        printf("        u2'(t) = 2u1(t)                 u2(0) = 7\n");
        printf("        for which the exact solution is known to be:\n\n");
        printf("        u1(t) = 12exp(4t) - 2exp(-4t)\n");
        printf("        u2(t) =  6exp(4t) +  exp(-4t)\n\n");
        printf("        for     t = %7.3e to %7.3e, with %u steps\n\n",
                TSTART,TEND,STEPS);

        /*
                integrate the answer from t = 0 to t = 10 sec
                STEPS points.
        */

        init[0] = Y1ZERO;       /* set up initial condition matrix */
        init[1] = Y2ZERO;

        rk4n(fn,source,store,SYSIZE,TSTART,TEND,STEPS,init,tarray,yarray);
                        /* compute the answers for 1 function */

        /*  Print out solutions  */

        for(i=0;i<SYSIZE;i++)
                for(j=0;j<STEPS;j++)    /* print solution */
                        solutn(j,i);


        printf("\n\nEnd of execution\n\n");
}
```

End Listings

The purchase of your business computer system cost you thousands, maybe millions, of dollars.

For $14.97, Business Software can help you get what you paid for, and more.

Business Software magazine is a valuable resource that helps you squeeze every penny's worth of potential out of your computer system. It is written for the business person and decision maker who wants practical, "hands-on" software solutions to real problems, written in a straight-forward, jargon-free language.

Every month, **Business Software** brings you: case studies of business computer applications; company profiles, including who is using what software and why; software tutorials that explain what the manual left out; information on the latest products, as well as the longtime favorites, all from a business perspective.

To subscribe, please fill out this coupon and return it to:
Business Software, P.O. Box 27975, San Diego, CA 92128

☐ Yes! Please begin my subscription to **Business Software.**

Name ______________________________

Address ______________________________

City ______________________________

State ____________________ Zip ____________

☐ Bill me later ☐ $14.97 enclosed

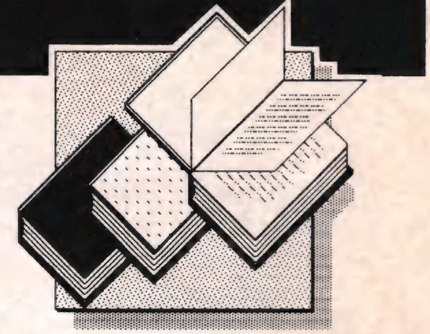
Please bill my ☐ VISA ☐ M/C ☐ Amer. Exp.

Card No. ______________________________

Expiration date ______________________________

Signature ______________________________

4001



CP/M Techniques

Ken Barbier
Published by Prentice-Hall, Inc.
225 pages
Reviewed by Dennis Cashton

If you are as much of a CP/M nut as I am, this book will become indispensable for you. In a very clear, consise, and friendly manner, Ken Barbier manages to discuss the real "guts" of CP/M. If you have worked with modifying your BIOS using assembly language under CP/M and have had to do anything that requires looking at the Digital Research documentation, your hair is probably lying in piles on the floor at the foot of your disk drives. This book takes the mystery out of all the vague topics about which "mother never told you."

I consider Mr. Barbier's previous work *CP/M Assembly Language Programming* a prerequisite for reading this book, but even if you have only a working knowledge of the mechanics of CP/M with an assembler, that should be good enough to get you through the book. In fact, even if you were a complete novice to CP/M, you'd just about be able to read and understand *CP/M Techniques*. (The "novice" level of CP/M is obtained by reading and not understanding the Digital Research documentation provided.)

The book starts with a general overview of a computer, what constitutes hardware, software, and firmware, and what constitutes an application program. Barbier follows this with a brief glance at the function of an operating system, and then jumps directly into CP/M. From this point on, the book is packed with sample programs and subroutines, each of which is immediately useful as well as being an excellent example of good programming practice. In fact, while reading the book, I had to run to the computer and enter each of the programs, just to make sure that they really do work. They do.

He goes on to a discussion of the medium of floppy disks. This section covers everything from history to current standards, and directions for the future (plus some more sample programs). Finally, he plunges head first into the part of CP/M that scares the most people: the BIOS. All the information and techniques illustrated in the prior chapters are brought to bear on the task of creating a customized BIOS for a user's own hardware configuration. What makes this section specially clear is the use of real-life examples. The book is full of them. They make such as abstract concepts like the IOBYTE much easier to understnd, mostly because you can see a real case of how and *why* each is used.

After you finish this book, it no longer seems like a monumental effort to get your CP/M to work with multiple printers, or different disk drives, or even a hard disk drive. You walk away from reading this book with a feeling that you can now conquer the (CP/M) world.

The book includes two appendices. The first is the ASCII code with mnemonics, keystrokes, hex, and function. The second is a listing of all the general purpose subroutines printed in the book, and a brief synopsis of their function.

In summary, if you have ever had a good mystery novel that you just couldn't put down, then you will understand the feeling you get from reading this book. It's so full of useful and concisely presented information that it is a pleasure to read. Even if you are not interested in customizing your own BIOS, this book is worth having for the programming techniques alone. If, on the other hand, you have the nerve to want to mess with your own hardware and BIOS, you'll find this book an invaluable tool.

The RS-232 Solution

Joe Campbell
Published by Sybex, Inc.
$16.95, 225 pages, illustrated, paperback
Reviewed by Dennis Cashton

Here's the plot: You get your new printer, the one that will do everything you ever wanted in a printer and more: high speed dot-matrix "correspondence quality" graphics with a 400K buffer. You assume that because the printer is "RS-232 compatible" you can plug it into the RS-232 port on the back of your computer, and everything will work fine.

Rushing home from the store with the printer, you tear the packing material off like it's wrapping on a box under the Christmas tree, glance at the cover of the instruction book, and lightly toss it aside. You don't need to look at it because "if it's RS-232, all I have to do is plug it in!" *Wrong!* You hook up the cable you bought with it (that the store owner said was a "standard" cable). If you're really lucky, the printer and the computer both have female connectors, because the cable you bought has male connectors on both ends. You turn on the computer. That part you've done a hundred times before, and you get no surprises. You turn on the printer, and the power light comes on, the printhead jumps a little, and the motor whirs. So far, so good.

You boot up CP/M and get no surprises there. With you heart in your throat, you hit control-p to toggle the printer on, and *zappo!* No more CP/M prompt! The keyboard is dead!!! What happened? Or better yet (more mystifying anyway) you get the prompt

back but you try to print something and nothing comes out. Is it the cable? The baud rate? Is the printer no good? What do you do now?

Well, this is a good time to look at the instructions book for the printer, the computer, and the operating system. Barring all complications, chances are the manufacturers of your printer and computer haven't totally lied about their RS-232 compatibility, but there are many ways to stretch the issue.

The RS-232 Solution will help anyone to connect any two RS-232 devices. It's the kind of reference book many of us need—there are, however, a few ommissions and discrepancies in this one. Campbell is careful to explain everything there is to know about RS-232, including all the buzzwords; unfortunately, there are some gaps in relating the buzzwords to the real signals and their meanings.

The author wisely sets up a convention in labeling his data-flow diagrams. He starts out writing the names of inputs in lowercase letters, marking them with a ?, and writing outputs in uppercase letters, marking them with a ! . On the very next page (and several other times in the book), he violates that convention. Although a minor inconvenience, it illustrates the nature of some of the book's flaws. I got the feeling that the book needed a technical wizard as an editor.

In spite of this minor beef, this book contains a wealth of knowledge about interfacing RS-232 devices. A set of instructions is developed that, if followed, will allow any real RS-232 device to connect to any other. The book is structured well, and can take even the total novice through the logic and signals of the RS-232 standard.

In explaining each one of the signal lines, the author uses many illustrations that make it easy to understand who's doing the talking and who's doing the listening. He also uses (usually humorous) cartoons illustrating excellent analogies to the purpose and functions of the handshaking lines, making these abstract concepts clear. There is even a chapter explaining the UART and how it functions in a serial interface. The book discusses signaling buffer full, pausing the transmitter and receiver, and generally what handshaking is all about. There is even some mention of software handshaking, but since this book deals mostly with the hardware end of things, it is understandably brief.

Eventually the book plunges the reader, who is armed only with the knowledge and theories gained from the previous chapters, into the real world of finicky devices. Topics with hobbyist-sounding connotations like "tricking the interface" come up, but as the reader soon finds out, this sort of fiddling is necessary in the real world. At this point in the book, you get the facts of "what happens if I don't do this." Once you get through the next couple of chapters, you are set to try things out on a real device.

In fact, one chapter discusses a simple tool that will make it easier to start out. I'm sure that many people have considered buying one of the "breakout boxes" on the market that allows them to configure a cable that will make their printer (or modem) work with their computer. The author describes here how to build, at a savings of over $100, a simple tool that performs the same function. I made one out of "junk box" parts in a little under 15 minutes, but if you had to buy all the parts new, it would still cost less than $20.

After this, you get to put your hands on your devices, and let the sparks fly. Actually, Campbell is careful to point out what the RS-232 standard says on that subject. According to the standard, any interface line must be capable of withstanding a direct connection to any other line without damage to the interface or the equipment. So there is really no danger of sparks. With this knowledge firmly in hand, we march on to learn and practice the interfacing technique.

Steps are numbered 1 through 5 and must be completed in order. They are:
1) Set baud rate
2) Ascertain sex of the equipment
3) Satisfy device control logic
4) Locate the handshaking
5) Specify the cable
As we learn later, some of these steps get combined, but they are still done in this general order. Each one of the steps is explained in detail, and you may now go happily on your way, connecting DTE's to DCE's all day long.

A latter section in the book takes you step-by-step through five actual case studies. They range from the simplest possible to the very tricky and difficult, but each one serves as an excellent example of all that has come before.

Basically, in spite of some of the confusion due to inconsistencies and unconnected concepts, this book should be on the shelves and workbenches of anyone who deals with these "universal" RS-232 devices. You could get the information in this book elsewhere, but only with much effort. I personally expect the pages of my copy of this book to become dog-eared and dirty.

The IBM PC Connection

James W. Coffron
Sybex Computer Books, 1984
$16.96, 260 pages
Reviewed by Eunice B. Ordman

The *IBM PC Connection* is a book for those with an elementary knowledge of BASIC (the IBM version of Microsoft BASIC) and no knowledge of electronics. It is a book for hobbyists, not a textbook. The author takes the reader step-by-step through both software and hardware aspects of interfacing the IBM PC. The uninitiated should be able to follow the author's instructions but may not understand much of what he is doing, particularly at first—Coffron tends to explain the functions of most of the components, without explaining how they work.

Readers will learn how to do sophisticated things such as building an alarm system that tells which switch set off the alarm. Other topics include computer speech output, temperature measurement with the computer using an integrated circuit as input, computer control of a variable speed motor, and analog-to-digital and digital-to-analog conversion.

For the uninitiated, Coffron discusses a preassembled input/output board and then progressing to more complex topics, he gives diagrams of circuits the reader can build. He does not present the reader with a huge complicated circuit, but introduces the parts of the circuit gradually—it takes perhaps four diagrams to present a given circuit. He even includes tips on reading schematic diagrams. The reader is on his own when it comes to soldering, wire wrapping, and so forth, but such information is widely available. The appendix gives the necessary manufacturer's specifications for each integrated circuit used and the name of a supplier for each type of part used. (The suppliers on his list whom I've used have very reasonable prices.)

Coffron tells how to revise the circuits and programs for different circumstances. With a few more sentences, he could have given an explanation of address decoding, but instead, Coffron contents himself with showing circuits that use certain port addresses, using those addresses in his programs. He gives a diagram of the IBM PC Input/Output Connector, showing the abbreviations for all 62 signals, but he does not tell what many of the abbreviations stand for.

I think the average hobbyist will be glad that the author did not confuse him with unnecessary details. I myself would have preferred a bit more explanation, however.

Advanced Pascal Programming Techniques

Paul A. Sand
Osborne/McGraw Hill, 1984
$14.95, 370 pages
Reviewed by Bob Langevin

If L-Name is the name of a computer language, your favorite bookseller's shelves are filled with books with titles such as *Introduction to L-Name*, *Easy L-Name*, and *L-Name for Beginners*. When, you wonder, will you get to the really advanced stuff? Sand's new book, *Advanced Pascal Programming Techniques* (APPT), is one of the few available Pascal texts that will take you well past the introductory level.

Overview

Sands correctly observes that most introductions to a computer language provide numerous small examples to illustrate each of the language's features but few texts, if any, show how these features are used in building substantial and meaningful programs. His book is a serious attempt to remedy this defect.

In its introductory chapter the book discusses "What is a Good Program"; the remaining seven chapters of the book are a series of detailed case studies, each illustrating important design concepts. The chapter titles give a good indication of the material covered: "CRT Techniques"; "Interactive Input"; "Crunching Numbers—A General Purpose Calculator"; "Text File Tools"; "Games and Strategy"; "Simulation and Animation"; "The Plane Truth—An Electronic Worksheet".

Because the Apple II version of UCSD Pascal was used to develop the programs in the book, the appendix is especially useful—it discusses the problems of program portability and shows how features unique to UCSD Pascal and the Apple II can be implemented in the context of other Pascal compilers on other machines.

Features

The first chapter discusses the features that make a good program, viewed from both the user's and the programmer's point of view, an important distinction that is often not made. From the user's viewpoint, the important features are usefulness, ease of use, efficiency, flexibility, reliability, and suitability; from the programmer's, they are readability, portability, clarity, and modularity. Sand emphasizes the need to recognize that, during its lifetime, a real program will be repeatedly fixed, modified, and enhanced—often by someone who did not design or implement the original. The succeeding chapters show, by example, how the user's needs can be met and the programmer's objectives achieved.

It is noteworthy that the design of the applications in each of the case studies has been carefully crafted so that procedures and functions developed in one chapter become useful tools in the design and implementation of the case studies in succeeding chapters. There is a definite flavor of the Unix design and development philosophy here.

In the second chapter, on CRT techniques, Sand shows how to write a Pascal procedure to manage the cursor and screen control features of a CRT. These are used in implementing a simple program that creates, displays, and solves mazes. This program is of little inherent interest, and the brief discus-

sion of CRT management procedures in the beginning of the chapter could have better appeared as an introduction to the next chapter of interactive input.

Unquestionably, the effective management of user interaction with a program is of critical importance in achieving user satisfaction and controlling input errors of all kinds. Sand, in Chapter 3, provides a good discussion of interactive input and error control in the course of developing "gaslog," an application program that maintains and reports gasoline usage data for an automobile. Surprisingly, this deceptively simple program provides the motivation for developing procedures for string input and string manipulation, boolean and fixed-point numeric input, date handling, and a variety of type transformers to convert between string and various numeric types.

In Chapter 4 Sand uses the development of a general-purpose calculator as a vehicle for addressing error control, a data structure for "extended" real numbers, a simple parser of user input, and functions to convert back and forth between strings and the "extended" reals that Sand has devised.

Next Sand discusses text file tool. These include tools to open or close an existing text file to read a string from or write a string to a text file, to create a new text file for output, and to get a text filename from the user. These functions are used to develop a simple file copy program and, much more interestingly, a rather elegant program for printing text files. The latter offers the opportunity for a good discussion of the use of escape sequences for initializing a printer and the incorporation of procedures for changing print parameters to control the format of the output.

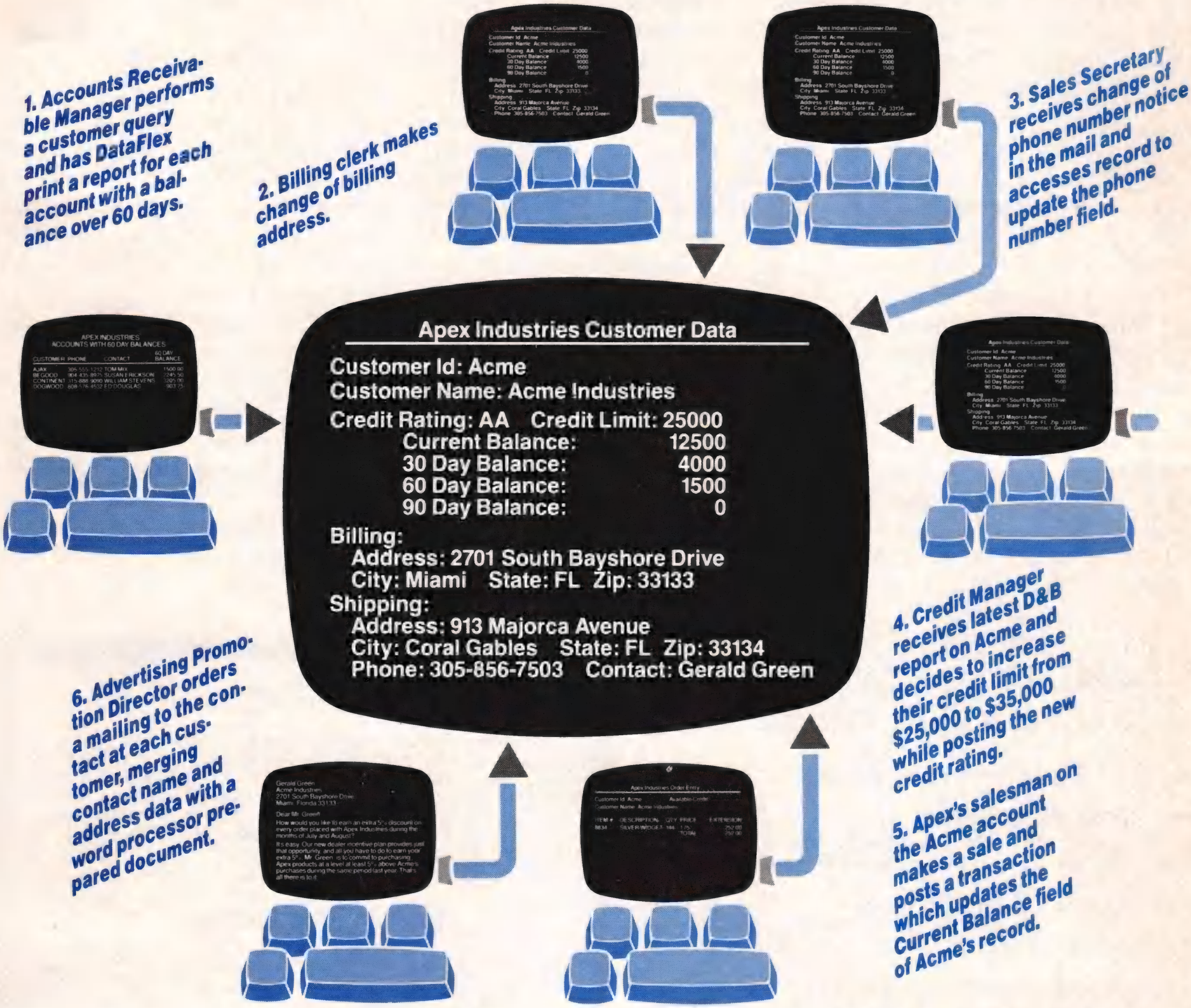
The print program also supports chaining of files to be printed. Aside from its utility as a learning device, this print program is a useful tool for any program developer. In addition if used with a good full-screen text editor, the combination makes a serviceable word-processing package.

Chapter 6 is devoted entirely to the design and implementation of a program to play Reversi—with a brief side trip to describe Apple II graphics capabilities. While this has some interest if you happen to want a program to play Reversi, the design considerations are so unique to the structure of this particular game and game board that you can neither generalize nor extend considerations of Reversi's appropriate data structures and procedures for moving pieces and position evaluation to other games. If you are not a Reversi fan, you can skip this chapter without losing much.

Next, simulation and animation are illustrated by designing and implementing two simulations: "bounder," which simulates the motion of several balls moving inside a box under the influence of a constant acceleration field, and "isaac," which uses many of the same program components and simulates the motion of several balls

ALL AT ONCE!

AND NEVER A "LOCKED OUT" USER!



DataFlex is the only application development database which ***automatically*** gives you true multi-user capabilities. Other systems can lock you out of records or entire files for the full time they are being used by someone else. DataFlex, however, locks only the data being changed, and ***only*** during the micro-seconds it takes to actually write it to the file! The updated record is then immediately available. The number of users who can access, and change, records at the same time is limited only by the number of terminals on your system or network. Call or write today for all the details on DataFlex...the true multi-user database.

DATAFLEX™

DATA ACCESS CORPORATION
8525 SW 129 Terrace, Miami, FL 33156 (305) 238-0012
Telex 469021 DATA ACCESS CI

Compatible with CP/M-80, MSDOS networks, MP/M-86, Novell Sharenet, PC-Net, DMS Hi-net, TurboDOS multi-user, Molecular N-Star, Televideo MmmOST, Action DPC/OS, IBM PC w/Corvus, OMNINET, 3Com EtherSeries and Micromation M/NET.

Circle no. **20** on reader service card.

moving under their mutual gravitational attraction. Several sets of initial conditions are provided for "isaac" that result in simulations of earth-sun, earth-moon-sun, and Lagrangian systems. The displays associated with both simulations depend entirely on the graphics display capabilities of the Apple II and would require major revision to accommodate the graphic characteristics of other computer systems.

Sand's book concludes with the development of "pascalc," a basic but usable spreadsheet program. Although this program does not approach the sophisticated capabilities of commercially available spreadsheet programs, it affords a useful study of Pascal procedures for dynamic memory management. In addition, and since any useful spreadsheet has dimensions substantially larger than the available screen area, "pascalc" includes the development of effective screen management procedures. As in the preceding chapters, Sand has taken some care to design "pascalc" so that it makes substantial use of procedures and functions developed earlier in the book.

The book is strengthened by the inclusion of suggestions and recommended reading at the end of each chapter. The suggestions, some of which are quite challenging (and time-consuming), address possible and desirable enhancements to each chapter's program. The references included in the recommended reading are usefully annotated to indicate their relevance to the text—a uniformly desirable procedure.

Evaluation

There is no doubt that you can improve prove your programming skills by careful study of Sand's book, especially if you actually work through in detail the sections of interest to you. His book failed, however, to meet my expectations in several important respects.

First of all, the book is too "even." Sand gives you no indication whatsoever of the relative importance or general utility of any of the ideas, procedures, or functions discussed. A twelve-line procedure to send a termination sequence to a printer, for example, receives neither more nor less emphasis than "stor," which converts a string into a real number.

Second, because the book is structured around explicit case studies, related topics end up scattered through several chapters. By way of example, various functions to convert between strings and numeric data types appear in chapters 3, 4, 6, 8, and 9. As a result, the basic characteristics of such type conversions are never developed and presented. Data structures likewise "pop up" when they are required, so that an organized view of data structures and their relationships is entirely absent.

Finally and again due to its structure, the book isn't very suitable for reference purposes—you can't really look anything up in it. As indicated earlier, the table of contents is only eleven lines long, so it isn't of much help. The index isn't much better, particularly because most index entries reference data structures, procedures, and functions that are unique to the book's case studies. When the entries do purport to be more general, they are often nearly useless; for example, under the major heading of Algorithms the only subheadings that appear are Minimax and Shell sort!

In summary, *Advanced Pascal Programming Techniques* does develop numerous useful tools that are applied to the design and implementation of substantial programs. It does not, however, live up to its title. Techniques, as such, are never really discussed. You won't find in the book an organized presentation of typical "technique" topics such as screen design, error trapping, algorithm efficiency, program efficiency, data structures, modularity, design for maintainability, program testing, and documentation. If you prefer a strong technique orientation, you would do much better with a book such as *Advanced Programming and Problem Solving with Pascal* (Schneider and Bruell, John Wiley & Sons, 1981)—in my opinion, the best text available on advanced programming techniques.

Computer Algebra

V.162 of Lecture Notes in Computer Science

Edited by J. A. van Hulzen
Springer-Verlag
305 pages

Reviewed by Morton F. Kaplon

This volume, No. 162 in the Springer-Verlag series *Lecture Notes in Computer Science*, is the third one dedicated to a computer algebra conference. The conference, EUROCAL '83, was held at the Kingston Polytechnic, Surrey, England, from 28 March, 1983 to 30 March, 1983. EUROCAL represents the EUROpean Computer ALgebra Community. The conference was organized under the responsibility of SAME (Symbolic and Algebraic Manipulation in Europe) and in cooperation with SIGSAM (Special Interest Group on Symbolic and Algebraic Manipulation) and with the official approval of ACM.

A natural question is "What is computer algebra?" A response is best given by quoting from the concluding statements of Professor van Hulzen's introduction:

"Many of the conference participants recognized that publication of real lecture notes about fundamental aspects and use of computer algebra, for instance in this Series, might largely contribute in establishing the user community and the more general interest computer algebra deserves, at least according to its adepts. This will certainly contribute to a *communis opinio*, and probably also to a 'definition'."

Does that leave you puzzled? This volume presents 27 research papers, organized in seven categories as follows: Algorithms 1 — Miscellaneous; Applications — Miscellaneous; Systems and Language Features; Algorithms 2 — Polynomial Ideal Bases; Algorithms 3 — Computational Number Theory; Algorithms 4 — Factorization; and System Oriented Applications. This volume and the papers presented are not for the novice and do not constitude easy reading. The range of material covered is quite broad, covering the spectrum from aspects of pure mathematics to the realization of programs in terms of specific hardware.

The first paper, entitled "Integration—What do We Want from the Theory?", reflects one end of that spectrum. The author, in discussing the state of the theory relating to the

integration of algebraic functions, proves an existence theorem using a non-constructive proof—certainly, not a very useful result for those interested in writing programs. At the other end we have "Implementing REDUCE on a Microcomputer" (this is not a recipe) and "The Design of MAPLE: A Compact, Portable, and Powerful Computer Algebra System." The latter will yield to the programmatically inclined a sense of the aims of *Computer Algebra* as reflected in software.

Do not be misled, however. As noted, the range of *Computer Algebra* is broad, but that range does include significant aspects relating to artificial intelligence. There also are articles that are certainly germane to developments in microcomputer software and perhaps even to hardware at a very basic level. These include, among others, "A Knowledge-Based Approach to User-Friendliness in Symbolic Computing," "Computer Algebra and VLSI," and "The Bath Concurrent LISP Machine." If you want to get a feel for what is going on at the frontier of this interesting and important field, a few dedicated hours of selective reading from this volume may prove quite informative and even potentially useful. And if you know LISP, you will even occasionally feel at home.

New Books

In addition to books we formally review, we see lots of titles that we think you might as least want to know about. We will from time to time provide a list of such books along with a brief impression or description. Here is our first batch.

Ada—An Advanced Introduction Including Reference Manual for the Ada Programming Language
Narain Gehani
Prentice-Hall, Englewood Cliffs,New Jersey, 1984.
$19.95, 291 pages
One thick book.

Advanced Programming in Microsoft BASIC
Gabriel Cuellar
Reston Publishing, Reston, Virginia, 1984.
$16.95, 152 pages
A "second" book on BASIC. Contains programming tools and hints.

Directory of Public Domain (And User-Supported) Software For the IBM Personal Computer
PC SIG
PC SIG, Santa Clara, California, 1984.
$4.95, 109 pages
The PC Software Interest Group's directory.

Free Software For the IBM PC
Bertram Gader and Manuel V. Nodar
Warner, New York, 1984.
$8.95, 462 pages
Software on 45 electronic bulletin boards.

Interactive Programming Environments
David R. Barstow, Howard E. Shrobe, and Erik Sandewall, eds.
McGraw-Hill, New York, 1984.
$34.95, 570 pages
Adele Goldberg on Object languages, Brian Kernighan on Unix, Terry Winograd looking beyond programming languages.

Invitation to Ada
Condensed Version
Harry Katzan, Jr.
$14.95, 166 pages
Petrocelli, New York, 1984.
Hospitable Harry Katzan, Jr., has written a series of "Invitiation to—" books for Petrocelli. Here, he invites the almost-beginner to sample the rudiments of the language they speak at the Department of Defense.

Learning LISP
Jeff Shrager, Steve Bagley, Steware Schiffman and Steve Cherry
Prentice-Hall, Englewood Cliffs, New Jersey, 1984.
$14.95, 199 pages
Comes with a LISP disk for the Apple.

On Conceptual Modeling
Perspectives from Artificial Intelligence, Databases, and Programming Languages
Topics in Information Systems series
Michael L. Brodie, John Mylopoulos, and Joachim W. Schmidt, eds.
Springer-Verlag, New York, 1984.
$29.25, 460 pages
How to encode knowledge. Papers presenting three approaches to a fundamental problem in the design of the commercial artificial intelligence packages called expert systems.

Pascal For Programmers
Olivier Lecarme and Jean-Louis Nebut
McGraw-Hill, New York, 1984.
$22.95, 267 pages
Presents ISO Standard Pascal. Assumes knowledge of at least one high-level language. The authors believe that Pascal enjoys the popularity it does partly because, when it was released, no agency, organization or vendor made any attempt to support it.

Programming In C
Stephen G. Kochan
Hayden, Hasbrouck Heights NJ, 1983.
$18.95, 365 pages
Attempts to speak both to complete novices and to experienced programmers.

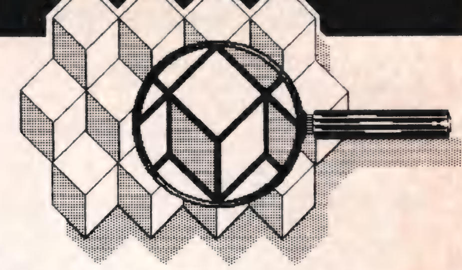
Programming With Structured Flowcharts
Krishna K. Agarwal
Petrocelli, New York, 1984.
$12.00, 166 pages
Why you should use those enclosed-space Nassi-Schneiderman flowcharts.

The Elements of Friendly Software Design
Paul Heckel
Warner, New York, 1984.
$8.95, 192 pages
Serialized in *InfoWorld* in 1982. How to program like D. W. Griffith. Heckel is nothing if not eclectic.

The IBM PC-DOS Handbook
Richard Allen King
Sybex, Berkeley, 1983.
$16.95, 288 pages
Includes appendices on the differences between PCDOS and MSDOS and among early version of PCDOS.

DDJ

OF INTEREST



by R.P. Sutherland

Software Tools

Borland (Turbo Pascal) International is now offering **Turbo Toolbox** to complement its Pascal compiler for Z80 and 8088/8086 microcomputers. Turbo Toolbox is a set of programming tools for data base, terminal installation, and sort applications. Two of the five tools, Turbo-ISAM (Index Sequential Access Method) and Quicksort, are available with commented source code. Turbo Toolbox is available for $49.95 from Borland International, 4113 Scotts Valley Drive, Scotts Valley, CA 95066. Reader Service No. 101.

Csharp Realtime Toolkit from the Systems Guild is a set of real-time, multitasking C programmer tools distributed in source code to allow user modification. Cgraph, for example, lets programmers write portable graphics programs and configure graphic system parameters by using C procedure calls. Csched (real-time scheduling of user procedures) creates a multitasking environment where each scheduled procedure can turn to completion unless interrupted by a procedure with a higher authority. Csharp tools are processor independent and run on 8- and 16-bit processors as well as on the entire PDP-11 family. Csharp tools will run under many operating systems, including Unix and RT11, and can also be imbedded in stand-alone software. The Csharp Realtime Toolkit is available under a single source license for $600 from Systems Guild, Inc., P.O. Box 1085, Cambridge, MA 02142. Reader Service No. 103.

C_to_dBASE, from Computer Innovations, provides the ability to write C applications for dBASE files. The $150 price tag (there are no additional royalty charges) includes the complete source code plus 70 functions that enable C programmers to perform operations on dBASE and index files. C_to_dBASE is available directly from Computer Innovations, Inc., 980 Shrewsbury Avenue, Suite R, Tinton Falls, NJ 07724. Reader Service No. 105

iLISP, from Computing Insights, is a new implementation of LISP for Z80 microcomputers. iLISP runs under CP/M 2.2 and is based on the LISP dialect developed by Gerry Sussman called SCHEME. One of the advanced LISP features includes a complete, extendable implementation of ELIZA, the famous psychotherapist parody. iLISP is available on both 8- and 5¼-inch disk formats (including Kaypro, Morrow, Zenith, and Osborne). The list price of $49.95 includes a 60-page introduction to iLISP programming. For more information write: Computing Insights, P.O. Box 4033, Madison, WI 53711. Reader Service No. 107.

Graphics

The STB **Graphix Plus II** is a video adaptor board for the IBM PC that supports both color and monochrome displays. I dropped one into my Eagle PC and now my Eagle flies "Flight Simulator." Three interesting extras are: (1) a 64K printer buffer; (2) "PC Accelerator," which supports print spooling and which offers a quick-start option that fools the PC into thinking only 64K of RAM is resident; and (3) an electronic RAM disk utility program. The package includes a 16-color driver for Lotus 1-2-3. The price is $495. For additional information contact STB Systems, Inc., 601 North Glenville Ave., Suite 125, Richardson, TX 75081. Reader Service No. 109.

Voices

ProTalker by Speech Ltd. provides S-100 systems and IBM PCs with voice output capability. Advanced users can interface ProTalker to most programming languages and applications because source code is provided. ProTalker is a digitizer/synthesizer that is switch selectable to rates of 4, 6, or 8 kHz. Adaptive delta pulse code modulation is used to reduce the size of digital recordings. Recordings are stored on disk until needed and can be accessed randomly under program control for play back. A telephone demo of ProTalker is running at (415) 858-2795. The price is under $350, available from Speech Ltd., 3790 El Camino Real, Suite 213, Palo Alto, CA 94306. Reader Service No. 111.

SynPhonix 100 is a speech synthesizer for the Apple II family. The board plugs directly into Apple II slots 1 – 7. The package includes a speech operating system on diskette. Users can generate speech and sound effects and incorporate them into software with standard BASIC statements. The SynPhonix 100 retails for $135 and is available from Arctic Technologies, 2234 Star Court, Auburn Heights, MI 48057. Reader Service No. 113.

The **VocaLink Speech Recognition Board** is a single-slot voice recognition board and software package for the IBM PC that allows users to operate off-the-shelf programs with up to 240 spoken commands. The SRB incorporates a high-speed, 16-bit Intel 80186 microprocessor to manage the complex operations required to accurately recognize discrete words/phrases spoken by a specific individual. Other hardware features include: the ASA-16 (a custom audio spectrum analysis

chip), 128K RAM, 32K EPROM, and the cabling required to connect a microphone, headset earphone, and voice synthesizer. The VocaLink Speech Recognition Board costs $1700 and is aimed at OEMs and sophisticated end-users. Inquiries should be addressed to Jim Bright, Interstate Voice Products, 1849 W. Sequoia Ave., Orange, CA 92668. **Reader Service No. 115.**

Macintosh

MacModula

Modula Corporation believes that, compared to Modula-2, using any other programming method is analogous to doing arithmetic in Roman numerals. If Modula-2 is to replace Pascal, then a Pascal-to-Modula-2 converter seems a sensible product. Modula Corporation has announced a range of software tools for use by developers of Modula-2 applications. A Modula-2 compiler, interpreter, and a **Pascal-to-Modula-2 source-code converter** are available for Macintosh as well as for Apple's other machines (and IBM PCs and compatibles). The Macintosh versions feature enhanced, bit-mapped graphics support, which the company claims is superior to any Apple-supplied Macintosh products. Modula Corporation is at 950 N. University Ave., Provo, UT 84604. **Reader Service No. 117.**

The Ultimate Mr. Potato-Head

MicronEye for Macintosh is an imaging peripheral that allows the Macintosh to take pictures! The user can accept an image into MacPaint and then subject that image to all of the features of MacPaint. The possibilities for Mr. Potato-Head-like manipulation of one's friends or enemies are staggering. MicronEye is available for $395 from Micron Technology, Inc., Vision Systems Group, 2805 E. Columbia Road, Boise, ID 83706. **Reader Service No. 119.**

Mac Learns to Read

Oberon International has a Z80-based type-reader that is compatible with Macintosh. **Omni-Reader** can read four different typefaces and has the ability to learn others. The device allows one to read text into a computer at the rate of 80 characters per second. Omni-Reader is manufactured in the United Kingdom and retails for $500. Contact Oberon International, McArthur Plaza, Suite 630, LB48, 5525 McArthur Blvd, Irving, TX 75062. **Reader Service No. 121.**

Attention User Groups

If you would like your user group to be included in a ***National Directory of User Groups***, send a stamped, self addressed envelope to: Ken Ryder, P.O. Box 4102, Rome, NY 13440.

BLAISE PASCAL (1623-1662)

Chips Off the Old Block

Kids are fast unlearning passive (television addict) approaches to cathode ray tubes and discovering that monitors connected to microprocessors are devices to make dance.

Computers 'n Kids Adventure Fair is looking for games and educational software written by people up to 17 years old. Winners will have the chance to demonstrate their programs at the first annual Computers 'n Kids Adventure Fair, which will be held at the San Francisco Concourse (8th and Brannan) from October 18 – 21. For more information call (415) 848-6860.

Isaac Asimov has published a book for children called ***How Did We Find Out About Computers?*** Asimov places the invention of the computer in the context of history by tracing the development of man's ability to calculate from counting on fingers, to the abacus, to the slide rule, to the inventions of Pascal and Babbage. The illustrations by David Wool include Pascal (illustration center page), Jacquard's first loom, the Hollerith tabulator, Vannevar Bush's differential analyzer, and Aiken's Mark One. Asimov concludes this well-written and informative history with an affirmation of human intelligence (or is it a challenge to *DDJ* readers?): ". . . when I write a story, I write as fast as I can and put one word after another in just the right order until I am finished. But how can I tell a computer to do it? Even if I put a whole dictionary of words into its memory, how can I tell it what word to put first, and what second, and what third? How can I explain to it how to choose the order of words so that it can write a brand-new story, just the way I do it, when I don't know how I do it?" *How Did We Find Out About Computers?* is priced at $8.85 (ISBN 8027-6533-5) from the publisher, Walker and Company, 720 Fifth Avenue, New York, NY 10019. **Reader Service No. 123.**

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 198.**

ADVERTISER INDEX

We thought about calling it MacSimplex . . . after all it makes your IBM®PC behave like a Macintosh™ and much more . . .

and with over two years in the making, the Simplex Database Management System has features like 32-megabyte virtual memory and the most powerful networked/relational database in the microcomputer industry. Simplex was designed around how you think and the Macintosh way, so that you can use your favorite mouse to handle those mundane tasks like menu selection and data manipulation. And, if you don't have a mouse, you can use our keyboard mouse simulator, MouSim™.

Pop-up and pull-down menus, dialog and alert boxes are not just added features, they are the heart of the Simplex way. In addition, Simplex gives you both a software and a hardware floating point capability, each with 19-digit accuracy. It permits login, password, privilege, and can be used on a local area network. Simplex has full communications and a remote or local printer spooler. Above all, Simplex is modular and grows with you! Simplex also has a full-featured, English-like language which is simple to use.

You can't buy Simplex™, but it is now available as an integral part of *it's my* ***Business***™ and will be used by *it's my* ***Word***™, *it's my* ***Graphics***™, . . .

Businessmen! *it's my* ***Business*** will revolutionize the way that you handle your business. It saves time, money, and standardizes your system for all who use it. *it's my* ***Business*** comes with applications like accounting, interoffice or intraoffice mail, editing, invoicing, inventory managment, mail list, calendar, scheduler, forms and more. You can modify each of these to create applications specifically designed for you... maybe we should have called it "it's your Business".

Professionals! *it's my* ***Business*** has over 200 pages of examples and demonstrations to show you how to solve your everyday professional problems. And if these examples aren't enough, we give you a complimentary one-year subscription to Questalk™, our hands-on Simplex applications magazine.

System integrators and consultants, beware! If you are not using *it's my* ***Business*** with Simplex to solve your problems, don't be surprised when more novice programmers solve that complex math, industrial engineering, or business problem faster. We think that you can cut your concept-to-development time by an order of magnitude!

it's my ***Business*** (includes *it's my* ***Editor***) - $695.00
it's my ***Business*** Demo Disk - $20.00
it's my ***Editor*** - $100.00.

Quest Research software is available through your local computer store or through mail order from Quest Software Corporation at (205) 539-8086. 303 Williams Avenue, Huntsville, AL 35801.

Value added resellers and dealers please contact Quest Research, Incorporated at (800) 558-8088. 303 Williams Avenue, Huntsville, AL, 35801.

Quest™

Quest Research Inc.

This is THE PASCAL COMPILER You've Been Hearing About

"It's almost certainly better than IBM's Pascal for the PC... Recommended."
Jerry Pournelle
Byte, May 1984

$49.95

"If you don't have CP/M [for your Apple], Turbo Pascal is reason enough to buy it."
Cary Hara
Softalk Apple, May 1984

". . . an excellent product at an extraordinary price."
David D. Clark, *Dr. Dobb's Journal*, June 1984

And Now It's Even Better Than You've Heard!

- **Windowing (IBM PC, XT, jr. or true compatibles)**
- **Color, Sound and Graphics Support (IBM PC, XT, jr. or true compatibles)**
- **Optional 8087 Support *(available at an additional charge)***
- **Automatic Overlays**
- **A Full-Screen Editor that's even better than ever**
- **Full Heap Management—via dispose procedure**
- **Full Support of Operating System Facilities**
- **No license fees.** You can sell the programs you write with Turbo Pascal without extra cost.

Yes. We still include Microcalc . . . the sample spreadsheet written with Turbo Pascal. You can study the source code to learn how a spreadsheet is written . . . it's right on the disk.* And, if you're running Turbo Pascal with the 8087 option, you'll never have seen a spreadsheet calculate this fast before!

**Except Commodore 64 CP/M.*

Order Your Copy of TURBO PASCAL® VERSION 2.0 Today

For VISA and MasterCard orders call toll free: **1-800-255-8008**
In California: **1-800-742-1133**
(lines open 24 hrs, 7 days a week)
Dealer & Distributor Inquiries Welcome 408-438-8400

Choose One (please add $5.00 for shipping and handling for U.S. orders. Shipped UPS)

____ Turbo Pascal 2.0 $49.95 + $5.00
____ Turbo Pascal with 8087 support $89.95 + $5.00
____ Update (1.0 to 2.0) Must be accompanied by the original master $29.95 + $5.00
____ Update (1.0 to 8087) Must be accompanied by the original master $69.95 + $5.00

Check ____ Money Order ____
VISA ____ Master Card ____
Card #: ____
Exp. date: ____

BORLAND INTERNATIONAL
Borland International
4113 Scotts Valley Drive
Scotts Valley, California 95066
TELEX: 172373

My system is: 8 bit ____ 16 bit ____
Operating System: CP/M 80 ____
CP/M 86 ____ MS DOS ____ PC DOS ____
Computer: ____
Disk Format: ____
Please be sure model number & format are correct.
Name: ____
Address: ____
City/State/Zip: ____
Telephone: ____

California residents add 6% sales tax. Outside U.S.A. add $15.00 (If outside of U.S.A. payment must be by bank draft payable in the U.S. and in U.S. dollars.) Sorry, no C.O.D. or Purchase Orders. F11

Circle **no. 9** on reader service card.